

(51) International Patent Classification:
G06F 9/30 (2006.01)(21) International Application Number:
PCT/US2017/040509(22) International Filing Date:
30 June 2017 (30.06.2017)

(25) Filing Language: English

(26) Publication Language: English

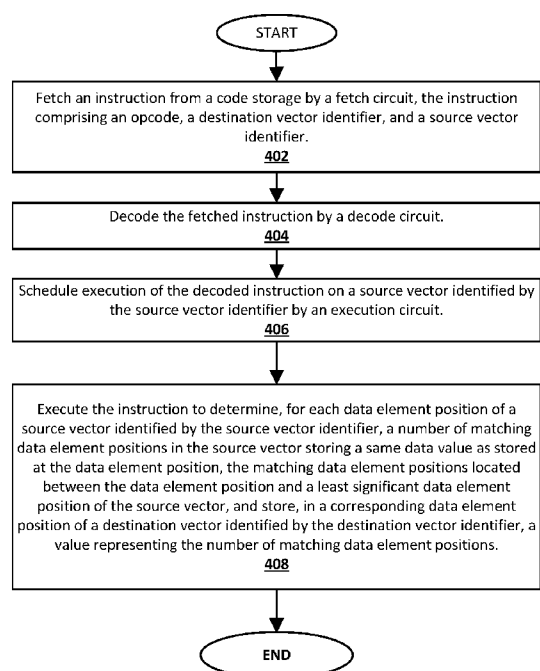
(71) Applicant: **INTEL CORPORATION** [US/US]; 2200
Mission College Boulevard, Santa Clara, California 95054
(US).(72) Inventors: **PLOTNIKOV, Mikhail**; Turgeneva Str. 30,
Nizhny Novgorod 603024 (RU). **HUGHES, Christopher
J.**; 3543 Druffel Pl., Santa Clara, California 95051 (US).
NARAIKIN, Andrey; Turgeneva Str. 30, Nizhny Nov-
gorod 603024 (RU).(74) Agent: **SELLARS, Eric S.** et al.; Nicholson De Vos Web-
ster & Elliott, LLP, 99 Almaden Boulevard, Ste. 710, San
Jose, California 95113 (US).(81) Designated States (unless otherwise indicated, for every
kind of national protection available): AE, AG, AL, AM,
AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ,
CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO,DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN,
HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP,
KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME,
MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ,
OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,
SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN,
TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.(84) Designated States (unless otherwise indicated, for every
kind of regional protection available): ARIPO (BW, GH,
GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ,
UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ,
TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK,
EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV,
MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM,
TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW,
KM, ML, MR, NE, SN, TD, TG).**Declarations under Rule 4.17:**

— of inventorship (Rule 4.17(iv))

Published:

— with international search report (Art. 21(3))

(54) Title: METHOD AND APPARATUS FOR VECTORIZING HISTOGRAM LOOPS

FIG. 4

(57) Abstract: Disclosed embodiments relate to a new instruction for detecting conflicts in a set of vector elements and determining a number of instances of each distinct data value within the vector. A system includes circuits to fetch, decode, and execute an instruction that includes an opcode, a destination vector identifier, a source vector identifier, and an immediate value, wherein the execution circuit is to, for each data element position of a source vector, determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and store in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.



METHOD AND APPARATUS FOR VECTORIZING HISTOGRAM LOOPS**FIELD OF INVENTION**

[0001] The field of invention relates generally to computer processor architecture. More specifically, the invention relates to an instruction for detecting conflicts within a vector of data values and determining a number of instances of each distinct data value within the vector of data values.

BACKGROUND

[0002] An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). It should be noted that the term “instruction” generally refers herein to macro-instructions – that is instructions that are provided to the processor for execution – as opposed to micro-instructions or micro-ops – that is the result of a processor’s decoder decoding macro-instructions. The micro-instructions or micro-ops can be configured to instruct an execution unit on the processor to perform operations to implement the logic associated with the macro-instruction.

[0003] The ISA is distinguished from the microarchitecture, which is the set of processor design techniques used to implement the instruction set. Processors with different microarchitectures can share a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. For example, the same register architecture of the ISA may be implemented in different ways in different microarchitectures using well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement register file). Unless otherwise specified, the phrases register architecture, register file, and register are used herein to refer to that which is visible to the software/programmer and the manner in which instructions specify registers. Where a

distinction is required, the adjective “logical,” “architectural,” or “software visible” will be used to indicate registers/files in the register architecture, while different adjectives will be used to designate registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

[0004] An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down through the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format’s fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. A given instruction is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and specifies the operation and the operands. An instruction stream is a specific sequence of instructions, where each instruction in the sequence is an occurrence of an instruction in an instruction format (and, if defined, a given one of the instruction templates of that instruction format).

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

[0006] Figure 1 illustrates an exemplary execution of a conflict detection instruction, according to some embodiments;

[0007] Figure 2 is a block diagram illustrating processing components for executing an instruction to detect conflicts within a vector of data values and to determine a number of instances of each distinct data value within the vector according to one embodiment;

[0008] Figure 3 is a block diagram illustrating example formats of a generic vector conflict detection and data value count instruction, according to some embodiments;

[0009] **Figure 4** is a flow diagram of a process to be performed by a processor to execute a vector conflict detection and data value count instruction, according to some embodiments;

[0010] **Figure 5A** is pseudocode illustrating execution of a vector conflict detection and data value count instruction, according to an embodiment;

[0011] **Figure 5B** includes diagrams illustrating execution of a vector conflict detection and data value count instruction, according to some embodiments;

[0012] **Figure 5C** is pseudocode and a diagram illustrating execution of an alternative vector conflict detection and data value count instruction, according to an embodiment;

[0013] **Figures 6A-6B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

[0014] **Figure 6A** is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention;

[0015] **Figure 6B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention;

[0016] **Figure 7A** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention;

[0017] **Figure 7B** is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the full opcode field 674 according to one embodiment of the invention;

[0018] **Figure 7C** is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the register index field 644 according to one embodiment of the invention;

[0019] **Figure 7D** is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the augmentation operation field 650 according to one embodiment of the invention;

[0020] **Figure 8** is a block diagram of a register architecture 800 according to one embodiment of the invention;

[0021] **Figure 9A** is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention;

[0022] **Figure 9B** is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention;

[0023] **Figures 10A-10B** illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip;

[0024] **Figure 10A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1002 and with its local subset of the Level 2 (L2) cache 1004, according to embodiments of the invention;

[0025] **Figure 10B** is an expanded view of part of the processor core in **Figure 10A** according to embodiments of the invention;

[0026] **Figure 11** is a block diagram of a processor 1100 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention;

[0027] **Figures 12-16** are block diagrams of exemplary computer architectures;

[0028] **Figure 12** shown a block diagram of a system in accordance with one embodiment of the present invention;

[0029] **Figure 13** is a block diagram of a first more specific exemplary system in accordance with an embodiment of the present invention;

[0030] **Figure 14** is a block diagram of a second more specific exemplary system in accordance with an embodiment of the present invention;

[0031] **Figure 15** is a block diagram of a System-on-a-Chip (SoC) in accordance with an embodiment of the present invention; and

[0032] **Figure 16** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention.

DETAILED DESCRIPTION

[0033] In the following description, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific

details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0034] References in the specification to “one embodiment,” “an embodiment,” “an example embodiment,” etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0035] “Sparse updates” are an important algorithmic pattern for which vectorization can be beneficial. To process a sparse update pattern, a read-modify-write operation may be performed on an indirectly addressed memory location (e.g., load $A[B[i]]$, add a computed value to it, and store the value back in $A[B[i]]$). Vectorizing this type of operation typically involves performing a gather-modify-scatter operation. By way of example, such an operation may involve performing 16 indirect loads of the form $A[B[i]]$ for 16 consecutive values of i via a gather operation, performing a single instruction multiple data (SIMD) computation, and scattering the new values back to memory. However, this vectorization assumes that a single gather/scatter instruction accesses each memory location no more than once. If, for example, two consecutive values of $B[i]$ are the same, then the read-modify-write for the second one is dependent on the first. As such, doing these simultaneously in a SIMD fashion violates these dependencies and can result in an incorrect result.

[0036] Another type of algorithmic pattern is a histogram computation. Like a sparse update pattern, a histogram pattern can be processed by a read-modify-write operation performed on indirectly addressed memory locations (e.g., load $A[B[i]]$, add a constant value to it, and store the value back in $A[B[i]]$). For example, consider the following pseudocode illustrating a histogram loop:

```
index_type *index;  
data_type *Out, C0;
```

```

for(i = 0; i < N; i++) {
    Out[index[i]] += C0;
}

```

[0037] As shown above, at each iteration of the loop, the pseudocode adds a constant value C0 to the value stored at Out[index[i]]. However, if multiple values of index[i] are the same, the operation can be performed more efficiently by determining how many times the constant value C0 is added at the same index, multiplying C0 by the determined number of times, and storing the resulting value to Out[i] only once.

[0038] One technique for vectorizing sparse updates and histogram operations involves using a conflict detection instruction, such as the VPCONFLICT instruction, to identify duplicate data elements of a vector. At a high level, a VPCONFLICT instruction compares the elements within a vector register (e.g., a register containing index values used to perform indirect updates to a vector of data values) and detects duplicate elements. More specifically, a VPCONFLICT instruction tests each element of its vector register input for equality with all earlier elements of that input (e.g., all elements closer to the least significant bit (LSB)) and outputs the results of these comparisons as a bit vector. In this manner, a VPCONFLICT conflict detection instruction provides a way to determine whether a given vector element has a data dependence involving other elements within the same vector.

[0039] **Figure 1** illustrates an exemplary execution of a VPCONFLICT instruction. As shown in **Figure 1**, the result of performing the VPCONFLICT instruction on an input vector 101 is an output bit vector 102, where each column of the output bit vector corresponds to a data element position of input vector 101 and indicates other data element positions that store a same value as stored at the data element position and are closer to the LSB. Bit positions of the column closer to the most significant bit (MSB) are filled with zeroes. For example, the column of output bit vector at element position 11 corresponds to the data element position 11 of the input vector 101 and indicates other data element positions (0, 1, 3, 6, and 7) of the input vector 101 that are closer to the LSB and store a same data value (9) as stored at the data element position 11. Bit positions of this column starting from 11 and up are filled with zeroes. Similarly, the column of output bit vector 102 at bit position 12

corresponds to the data element position 12 of the input vector 101 and indicates other data element positions (2, 4, and 5) of the input vector 101 that are closer to the LSB and store a same data value (5) as that stored at the data element position 12. Bit positions of this column starting from 12 and up are filled with zeroes.

[0040] One aspect of the VPCONFLICT instruction is that a full set of comparison results fit in a 512-bit wide vector register only if the size of the index is 32 or 64 bits. If each index is 16 bits wide, then there would be 32 bits of result for each index, but only 16 bits of space available in the register. The situation is even worse for 8 bit indexes. Currently, 16-bit and 8-bit indices are processed by upconverting the indices to 32-bit values, thereby losing 2x or 4x efficiency. Furthermore, the output of the VPCONFLICT instruction includes redundant information. For example, the *i*th row of the output bit vector has bits set indicating *all* matching elements closer to the LSB. As shown in **Figure 1**, the column corresponding to output element 6 indicates that it matches elements 3, 1, and 0, but the column corresponding to output element 3 also indicates that it matches elements 1 and 0.

[0041] Another aspect of the VPCONFLICT instruction is that it does not perform self-comparisons of data elements as part of the conflict check. For example, referring again to **Figure 1**, the column in the destination vector register corresponding to the element 11 indicates that it conflicts with elements 0, 1, 3, 6, and 7, but does not indicate that it conflicts with itself at element 11. Thus, if a separate instruction is used to count a total number of bits containing the value 1 for each data element position (e.g., VPOPCNT), the result indicates the number of instances of that data element minus one. For example, there are 5 bits containing the bit value 1 in the column corresponding to the element 11 in **Figure 1**, but there are 6 data elements in total containing the same data element value 9 in the source vector). This can complicate the computation of histogram patterns because histogram patterns typically add a constant value to each index position however many times the index position is present in a source index vector.

[0042] In some embodiments, a new instruction for use in computing histogram update patterns is disclosed. As described in more detail herein, the execution of the new instruction performs vector conflict detection and data value count operations in a single instruction to more efficiently compute histogram update patterns. Among other benefits,

the output values of the instruction are small enough to fit multiple data types in an output vector, including byte, word, dword, and qword values. Additionally, the instruction can include an immediate value that is added to the count result for each data element, thereby enabling the instruction to correctly indicate a number of times each distinct data element value is present in a source vector.

[0043] In some embodiments, an alternate version of the vector conflict and data value count instruction further includes a second source vector operand, where the instruction adds the result of the data value count to the second source vector. For a typical histogram computations, this can further optimize the implementation of histogram patterns in the common case where the output array contains integer values.

[0044] **Figure 2** is a block diagram of a processor to execute a vector conflict and data value count instruction, according to some embodiments. As shown, computing system 200 includes code storage 202, fetch circuit 204, decode circuit 206, execution circuit 208, registers 210, memory 212, and retire or commit circuit 214. An instruction is fetched by fetch circuit 204 from code storage 202, which may comprise a cache memory, an on-chip memory, a memory on the same die as the processor, an instruction register, a general register, or system memory, without limitation. In one embodiment, the instruction has a format similar to that of instructions 300A and 300B in **Figure 3**. After fetching the instruction from code storage 202, decode circuit 206 decodes the instruction, including by parsing the various fields of the instruction. After decoding the fetched instruction, execution circuit 208 executes the decoded instruction. When executing the instruction, execution circuit 208 may read data from and write data to registers 210 and memory 212. Registers 210 may include a data register, an instruction register, a vector register, a mask register, a general register, an on-chip memory, a memory on the same die as the processor, or a memory in the same package as the processor, without limitation. Memory 212 may include an on-chip memory, a memory on the same die as the processor, a memory in the same package as the processor, a cache memory, or system memory, without limitation. After the instruction has been executed, retire or commit circuit 214 may retire the instruction, ensuring that execution results are written to or have been written to their destinations, and freeing up or releasing resources for later use.

[0045] Figure 3 is a block diagram illustrating example formats of a generic vector conflict detection and data value count instruction, according to some embodiments. As shown, instruction 300A includes opcode 302, destination vector identifier 304, source vector identifier 306, and an immediate value 308. Destination vector identifier 304 identifies a memory location or a vector register in a register file to which to write data according to some embodiments. In some embodiments, a source vector identifier 306 identify a vector register in a register file, wherein the identified vector register serve as a source vector. Alternatively, the source vector can be taken from memory. An immediate value is a value stored as part of the instruction.

[0046] Instruction 300B includes an opcode 310, destination vector identifier 312, first source vector identifier 314, second source vector identifier 316, and an immediate value 318. Destination vector identifier 312 identifies a memory location or vector register in a register file to which to write data according to some embodiments. In some embodiments, a first source vector identifier 314 and a second source vector identifier 316 identifies a first and second vector register, respectively, in a register file, wherein the identified vector registers server as a source vector. Alternatively, one or both source operands can be taken from memory.

[0047] Figure 4 is a flow diagram of a process to be performed by a processor to execute a vector conflict and data value count instruction, according to some embodiments. After starting, the process at 402 fetches an instruction from a code storage by a fetch circuit, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value. At 404, a decode circuit decodes the fetched instruction. At 406, execution of the decoded instruction on a source vector identified by the source vector identifier by an execution circuit is scheduled.

[0048] At 408, the execution circuit determines, for each data element position of a source vector identified by the source vector identifier, a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and stores, in a

corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0049] In some embodiments, the matching data element positions located between the data element position and the least significant data element position of the source vector include the data element position. In this manner, the instruction includes a self-comparison of each data element position so that the count for each distinct data value indicates the total number of instances of the distinct data value. In one embodiment, the immediate value included in the instruction can be added to the data value count for each distinct data value so that the data value count reflects the total number of instances of each distinct data value.

[0050] In one embodiment, the source vector identifier is a first source vector identifier and the source vector is a first source vector, and the instruction further comprises a second source vector identifier. In this embodiment, storing the value representing the number of matching data element positions further includes adding the value to a corresponding data element position of a second source vector identified by the second source vector identifier. In some embodiments, the use of the immediate value and additional source vector can be implemented as extensions to a separate data value count instruction (e.g., the VPOPCNT instruction).

[0051] **Figures 5A-5C** include pseudocode and diagrams illustrating embodiments of a process to be performed by a processor to execute a vector conflict and data value count instruction, according to some embodiments.

[0052] Embodiments of the pseudocode listed in **Figures 5A and 5C** may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Some embodiments of pseudocode are to be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[0053] The pseudocode listed in **Figures 5A and 5C** may not by itself suffice to perform all aspects of an application or other software. The pseudocode listed in **Figures 5A and 5C** illustrate relevant aspects of embodiments disclosed herein. Additional software routines to

control inputs and outputs and other functionality are known to those of ordinary skill and may be used.

[0054] As shown in pseudocode 500 in **Figure 5A** and in the diagram 501 in **Figure 5B**, execution of a vector conflict and data value count instruction “VCONF_CNT” determines, for each data element position of a source vector identified by the source vector identifier (e.g., each data element position “src[i]”), a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector. For example, the loop “for (k=0; k<i; k++)” checks whether any of the data element positions “src[k]” to the right of the current data element position “src[i]” store a same data value as stored at position “src[k]” and, if so, increments the value n. An initial value for n is taken from an immediate source operand “imm8” (in the example of **Figure 5A**, imm8 might be initially set to 1). The pseudocode 500 further illustrates that the value n (indicating a number of data elements in the source vector to the right of src[i] storing the same value plus the immediate value) is stored in a corresponding data element position of a destination vector (“dest[i] = n”).

[0055] In **Figure 5B**, diagram 501 shows an example source vector 502 having eight data elements. Diagram 501 also shows a destination vector 503 storing values based on performance of a vector conflict detection and data value count instruction on the source vector 502. For example, the data element position 7 of the destination vector stores the value 5 indicating that there are 4 instances of the value stored at data element position 7 (9) to the right of the data element position 7 and plus 1 as an initial immediate value. The data element position 6 of the destination vector stores the value 4 to indicate that there are 3 instances of the value stored at data element position 6 (also 9) to the right of the data element position 6 and plus 1 as an initial immediate value. The data element position 5 stores the value 3 to indicate that there are 2 instances of the value stored at data element position 5 (5) to the right of the data element position 5 and plus 1 as an initial immediate value, and so forth. In the example of diagram 501, the immediate value 1 serves to include each data element position itself in the data value count, that is, the destination vector stores values indicating instances of the value stored at each data element position

to the right and including the data element position. As shown in **Figure 5B**, the destination vector 503 thus stores the partial sums of each data element value of the source vector 502 in the data elements of the destination vector 503 from right to left. The total population count in the source vector 502 for each data element value is stored at the left-most or most significant bit side of the destination vector 506 (e.g., the total population count for the distinct value 5 is stored at the left-most data element position 5 storing the value 5).

[0056] Diagram 504 shows an example of performing a similar vector conflict detection and data value count instruction (e.g., VLCONF_CNT). For example, the data element position 0 of the destination vector 506 stores the values 5 to indicate that there are 4 instances of the value stored at data element position 0 (9) to the left of the data element position 7 and plus 1 as an initial immediate value. Similarly, the data element position 2 of the destination vector 506 stores the value 3 to indicate that there are 2 instances of the value stored at data element position 2 (5) to the left of the data element position 2 and plus 1 as an initial immediate value. Thus, a total population count in the source vector 505 for each data element value is stored at the right-most or least significant bit side of the destination vector 506 (e.g., the total population count for the distinct value 5 is stored at the right-most data element position 2 storing the value 3).

[0057] Diagram 507 shows an example of performing a similar vector conflict detection and data value count instruction on data stored in big-endian format. For example, whereas the data elements of the source and destination vectors are stored in little-endian format in diagrams 501 and 504, the data elements of source vector 508 and destination vector 509 are stored in big-endian format. Similar to diagrams 501 and 504, either a right conflict or left conflict instruction can be performed on the data in source vector 508 depending on the implementation.

[0058] **Figure 5C** illustrates execution of an alternative vector conflict and data value count instruction "VCONF_CNT" comprising a destination vector identifier "dest," a first source vector identifier "src1," a second source vector identifier "src2," and an immediate value "imm8." As shown in pseudocode 510 and diagram 511, execution of the instruction determines, for each data element position of a first source vector identified by the first source vector identifier (e.g., each data element position "src1[i]"), a number of matching

data element positions in the first source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the first source vector. The number of matching data element positions is added to an initial value “n,” which is a sum of the corresponding data element position of a second source vector identified by the second source vector identifier and the immediate value imm8 (“ $n = \text{src2}[i] + \text{imm8}$ ”). For example, the loop “for ($k=0$; $k<i$; $k++$)” checks whether any of the data element positions “ $\text{src1}[k]$ ” to the right of the current data element position “ $\text{src1}[i]$ ” store a same data value as stored at position “ $\text{src1}[k]$ ” and, if so, increments the value n (initially set to the value stored at $\text{src2}[i]$ plus the immediate value). The pseudocode 510 further illustrates that the value n (indicating a number of data elements in the source vector to the right of “ $\text{src1}[i]$ ” storing the same value added to the initial value of n) is stored in a corresponding data element position of a destination vector (“ $\text{dest}[i] = n$ ”).

[0059] The diagram 511 illustrates an example of the alternative vector conflict and data value count instruction of pseudocode 510 executed on a first source vector 512 (storing a vector of index values) and a second source vector 513 (storing an initial vector of histogram values). For example, the data element position 7 of the destination vector 514 stores the value A+5. This value corresponds to the number of instances of the value stored at data element position 7 of the first source vector 512 to the right of data element position 7 (that is, 4 instances of the value 9) added to the initial value at the same data element position of the second source vector (A) and the immediate value 1. Similarly, the data element position 6 of the destination vector 512 stores the value A+4 corresponding to the 3 instances of the value stored at data element position 6 (also 9) to the right of the data element position 6 added to the initial value at the same data element position of the second source vector 511 and the immediate value 1, and so forth.

[0060] The pseudocode below illustrates use of a vector conflict detection and data value count instruction as described above to compute a typical histogram pattern:

[0061] For($i=0$; $i<N$; $i+=KL$) {
 vindex1 = VLOAD(index+i;)
 v1 = VGATHER(vindex1, Out);

```

    vres1 = VCONF_CNT(vindex1, 1); //imm8=1
    vres = VCVT_INT2FP(vres1);
    v1 = VFMA231(v1, vres, vC0); //vC0={C0, C0, C0, ... , C0}
    VSCATTER(Out, vindex1, v1);
}

```

[0062] As shown, the pseudocode performs a vector conflict and data value count instruction “VCONF_CNT” on the source vector “v1” with the immediate value 1 and stores the results in a results vector “vres1.” A VCVT_INT2FP operation is used to convert the integer values to floating-point values and a fused multiply-add instruction VFMA231 is used to multiply counted and converted to floating point value number of instances in the resulting vector “vres” by floating point constant C0 and add the result vector to the source vector “v1,” which holds data elements gathered from memory.

[0063] For the common case where the output vector contains integer values instead of floating-point values and the constant value to be added to identified index values is 1, a second version of the instruction can take advantage of a second source vector operand. In this example, the VCVT_INT2FP convert operation and fused multiply-add instruction “VFMA231” in the pseudocode above can be eliminated by providing the current contents of the histogram gathered from memory as the vector of starting values. The pseudocode below illustrates use of the second version of the instruction:

```

[0064] For(i=0; i<N; i+=KL) {
    vindex1 = VLOAD(index+i);
    v1 = VGATHER(vindex1, Out);
    v1 = VCONF_CNT(vindex1, v1);
    VSCATTER(Out, vindex1, v1);
}

```

[0065] As shown in the example pseudocode above, the vector “v1” is used to gather the current contents of the histogram in memory as the vector of starting values, and the VCONF_CNT instruction is used to add the results of the data value count to the initial source vector, thereby eliminating the VCVT_INT2FP and VFMA231 instructions from the previous pseudocode.

Instruction Sets

[0066] An instruction set may include one or more instruction formats. A given instruction format may define various fields (e.g., number of bits, location of bits) to specify, among other things, the operation to be performed (e.g., opcode) and the operand(s) on which that operation is to be performed and/or other data field(s) (e.g., mask). Some instruction formats are further broken down through the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands. A set of SIMD extensions referred to as the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developer's Manual, September 2014; and see Intel® Advanced Vector Extensions Programming Reference, October 2014).

Exemplary Instruction Formats

[0067] Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

Generic Vector Friendly Instruction Format

[0068] A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

[0069] **Figures 6A-6B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention.

Figure 6A is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while **Figure 6B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format 600 for which are defined class A and class B instruction templates, both of which include no memory access 605 instruction templates and memory access 620 instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

[0070] While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4 byte) or 64 bit (8 byte) data element widths (or sizes) (and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may support more, less and/or different vector operand sizes (e.g., 256 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte) data element widths).

[0071] The class A instruction templates in **Figure 6A** include: 1) within the no memory access 605 instruction templates there is shown a no memory access, full round control type

operation 610 instruction template and a no memory access, data transform type operation 615 instruction template; and 2) within the memory access 620 instruction templates there is shown a memory access, temporal 625 instruction template and a memory access, non-temporal 630 instruction template. The class B instruction templates in **Figure 6B** include: 1) within the no memory access 605 instruction templates there is shown a no memory access, write mask control, partial round control type operation 612 instruction template and a no memory access, write mask control, vsize type operation 617 instruction template; and 2) within the memory access 620 instruction templates there is shown a memory access, write mask control 627 instruction template.

[0072] The generic vector friendly instruction format 600 includes the following fields listed below in the order illustrated in **Figures 6A-6B**.

[0073] Format field 640 – a specific value (an instruction format identifier value) in this field uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in the vector friendly instruction format in instruction streams. As such, this field is optional in the sense that it is not needed for an instruction set that has only the generic vector friendly instruction format.

[0074] Base operation field 642 – its content distinguishes different base operations.

[0075] Register index field 644 – its content, directly or through address generation, specifies the locations of the source and destination operands, be they in registers or in memory. These include a sufficient number of bits to select N registers from a PxQ (e.g. 32x512, 16x128, 32x1024, 64x1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

[0076] Modifier field 646 – its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is, between no memory access 605 instruction templates and memory access 620 instruction templates. Memory access operations read and/or write to the memory

hierarchy (in some cases specifying the source and/or destination identifiers using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

[0077] Augmentation operation field 650 – its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field 668, an alpha field 652, and a beta field 654. The augmentation operation field 650 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[0078] Scale field 660 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

[0079] Displacement Field 662A– its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$).

[0080] Displacement Factor Field 662B (note that the juxtaposition of displacement field 662A directly over displacement factor field 662B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{scaled displacement}$). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field 674 (described later herein) and the data manipulation field 654C. The displacement field 662A and the displacement factor field 662B are optional in the sense that they are not used for the no memory access 605 instruction templates and/or different embodiments may implement only one or none of the two.

[0081] Data element width field 664 – its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other

embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported using some aspect of the opcodes.

[0082] Write mask field 670 – its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-write masking, while class B instruction templates support both merging- and zeroing-write masking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field 670 allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the write mask field's 670 content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's 670 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 670 content to directly specify the masking to be performed.

[0083] Immediate field 672 – its content allows for the specification of an immediate. This field is optional in the sense that is it not present in an implementation of the generic vector friendly format that does not support immediate and it is not present in instructions that do not use an immediate.

[0084] Class field 668 – its content distinguishes between different classes of instructions. With reference to **Figures 6A-6B**, the contents of this field select between class A and class B

instructions. In **Figures 6A-6B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 668A and class B 668B for the class field 668 respectively in **Figures 6A-6B**).

Instruction Templates of Class A

[0085] In the case of the non-memory access 605 instruction templates of class A, the alpha field 652 is interpreted as an RS field 652A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 652A.1 and data transform 652A.2 are respectively specified for the no memory access, round type operation 610 and the no memory access, data transform type operation 615 instruction templates), while the beta field 654 distinguishes which of the operations of the specified type is to be performed. In the no memory access 605 instruction templates, the scale field 660, the displacement field 662A, and the displacement scale field 662B are not present.

No-Memory Access Instruction Templates – Full Round Control Type Operation

[0086] In the no memory access, full round control type operation 610 instruction template, the beta field 654 is interpreted as a round control field 654A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 654A includes a suppress all floating point exceptions (SAE) field 656 and a round operation control field 658, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field 658).

[0087] SAE field 656 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 656 content indicates, suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

[0088] Round operation control field 658 – its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 658 allows for the changing of

the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 650 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

[0089] In the no memory access data transform type operation 615 instruction template, the beta field 654 is interpreted as a data transform field 654B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[0090] In the case of a memory access 620 instruction template of class A, the alpha field 652 is interpreted as an eviction hint field 652B, whose content distinguishes which one of the eviction hints is to be used (in **Figure 6A**, temporal 652B.1 and non-temporal 652B.2 are respectively specified for the memory access, temporal 625 instruction template and the memory access, non-temporal 630 instruction template), while the beta field 654 is interpreted as a data manipulation field 654C, whose content distinguishes which one of a number of data manipulation operations (also known as primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access 620 instruction templates include the scale field 660, and optionally the displacement field 662A or the displacement scale field 662B.

[0091] Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

Memory Access Instruction Templates – Temporal

[0092] Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates – Non-Temporal

[0093] Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

[0094] In the case of the instruction templates of class B, the alpha field 652 is interpreted as a write mask control (Z) field 652C, whose content distinguishes whether the write masking controlled by the write mask field 670 should be a merging or a zeroing.

[0095] In the case of the non-memory access 605 instruction templates of class B, part of the beta field 654 is interpreted as an RL field 657A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 657A.1 and vector length (VSIZE) 657A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 612 instruction template and the no memory access, write mask control, VSIZE type operation 617 instruction template), while the rest of the beta field 654 distinguishes which of the operations of the specified type is to be performed. In the no memory access 605 instruction templates, the scale field 660, the displacement field 662A, and the displacement scale field 662B are not present.

[0096] In the no memory access, write mask control, partial round control type operation 610 instruction template, the rest of the beta field 654 is interpreted as a round operation field 659A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

[0097] Round operation control field 659A – just as round operation control field 658, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 659A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for

specifying rounding modes, the round operation control field's 650 content overrides that register value.

[0098] In the no memory access, write mask control, VSIZE type operation 617 instruction template, the rest of the beta field 654 is interpreted as a vector length field 659B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[0099] In the case of a memory access 620 instruction template of class B, part of the beta field 654 is interpreted as a broadcast field 657B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 654 is interpreted the vector length field 659B. The memory access 620 instruction templates include the scale field 660, and optionally the displacement field 662A or the displacement scale field 662B.

[0100] With regard to the generic vector friendly instruction format 600, a full opcode field 674 is shown including the format field 640, the base operation field 642, and the data element width field 664. While one embodiment is shown where the full opcode field 674 includes all of these fields, the full opcode field 674 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 674 provides the operation code (opcode).

[0101] The augmentation operation field 650, the data element width field 664, and the write mask field 670 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

[0102] The combination of write mask field and data element width field create typed instructions in that they allow the mask to be applied based on different data element widths.

[0103] The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may

support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

Exemplary Specific Vector Friendly Instruction Format

[0104] **Figure 7A** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. **Figure 7A** shows a specific vector friendly instruction format 700 that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific vector friendly instruction format 700 may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode byte field, MOD R/M field, SIB field, displacement field, and immediate fields of the existing x86 instruction set with extensions. The fields from **Figure 6** into which the fields from **Figure 7A** map are illustrated.

[0105] It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 700 in the context of the generic vector friendly instruction format 600 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 700 except where claimed. For example, the generic vector friendly instruction format 600 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 700 is shown as having fields of specific sizes. By way of specific example, while the data element width field 664 is illustrated as a one bit field in the specific vector friendly instruction format 700, the invention is not so limited (that is, the generic vector friendly instruction format 600 contemplates other sizes of the data element width field 664).

[0106] The generic vector friendly instruction format 600 includes the following fields listed below in the order illustrated in **Figure 7A**.

[0107] EVEX Prefix (Bytes 0-3) 702 - is encoded in a four-byte form.

[0108] Format Field 640 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 640 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

[0109] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[0110] REX field 705 (EVEX Byte 1, bits [7-5]) – consists of a EVEX.R bit field (EVEX Byte 1, bit [7] – R), EVEX.X bit field (EVEX byte 1, bit [6] – X), and EVEX.B bit field (EVEX byte 1, bit [5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 911B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

[0111] REX' field 710 – this is the first part of the REX' field 710 and is the EVEX.R' bit field (EVEX Byte 1, bit [4] - R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the invention, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in

the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

[0112] Opcode map field 715 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[0113] Data element width field 664 (EVEX byte 2, bit [7] – W) - is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the datatype (either 32-bit data elements or 64-bit data elements).

[0114] EVEX.vvvv 720 (EVEX Byte 2, bits [6:3]-vvvv)- the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source vectors; 2) EVEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 911b. Thus, EVEX.vvvv field 720 encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers.

[0115] EVEX.U 668 Class field (EVEX byte 2, bit [2]-U) – If EVEX.U = 0, it indicates class A or EVEX.U0; if EVEX.U = 1, it indicates class B or EVEX.U1.

[0116] Prefix encoding field 725 (EVEX byte 2, bits [1:0]-pp) – provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for

different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

[0117] Alpha field 652 (EVEX byte 3, bit [7] – EH; also, known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also, illustrated with α) – as previously described, this field is context specific.

[0118] Beta field 654 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s₂₋₀, EVEX.r₂₋₀, EVEX.rr1, EVEX.LL0, EVEX.LLB; also, illustrated with $\beta\beta\beta$) – as previously described, this field is context specific.

[0119] REX' field 710 – this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3] - V') that may be used to encode either the upper 16 or lower 16 of the extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V', EVEX.vvvv.

[0120] Write mask field 670 (EVEX byte 3, bits [2:0]-kkk) – its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

[0121] Real Opcode Field 730 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

[0122] MOD R/M Field 740 (Byte 5) includes MOD field 742, Reg field 744, and R/M field 746. As previously described, the MOD field's 742 content distinguishes between memory access and non-memory access operations. The role of Reg field 744 can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 746 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

[0123] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 750 content is used for memory address generation. SIB.xxx 754 and SIB.bbb 756 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[0124] Displacement field 662A (Bytes 7-10) – when MOD field 742 contains 10, bytes 7-10 are the displacement field 662A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[0125] Displacement factor field 662B (Byte 7) – when MOD field 742 contains 01, byte 7 is the displacement factor field 662B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between -128 and 127 bytes' offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values -128, -64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field 662B is a reinterpretation of disp8; when using displacement factor field 662B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as $\text{disp8} * N$. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement assumes that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 662B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 662B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to $\text{disp8} * N$. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset). Immediate field 672 operates as previously described.

Full Opcode Field

[0126] Figure 7B is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the full opcode field 674 according to one embodiment of the invention. Specifically, the full opcode field 674 includes the format field 640, the base operation field 642, and the data element width (W) field 664. The base operation field 642 includes the prefix encoding field 725, the opcode map field 715, and the real opcode field 730.

Register Index Field

[0127] Figure 7C is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the register index field 644 according to one embodiment of the invention. Specifically, the register index field 644 includes the REX field 705, the REX' field 710, the MODR/M.reg field 744, the MODR/M.r/m field 746, the VVVV field 720, xxx field 754, and the bbb field 756.

Augmentation Operation Field

[0128] Figure 7D is a block diagram illustrating the fields of the specific vector friendly instruction format 700 that make up the augmentation operation field 650 according to one embodiment of the invention. When the class (U) field 668 contains 0, it signifies EVEX.U0 (class A 668A); when it contains 1, it signifies EVEX.U1 (class B 668B). When U=0 and the MOD field 742 contains 11 (signifying a no memory access operation), the alpha field 652 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 652A. When the rs field 652A contains a 1 (round 652A.1), the beta field 654 (EVEX byte 3, bits [6:4]– SSS) is interpreted as the round control field 654A. The round control field 654A includes a one bit SAE field 656 and a two bit round operation field 858. When the rs field 852A contains a 0 (data transform 852A.2), the beta field 854 (EVEX byte 3, bits [6:4]– SSS) is interpreted as a three bit data transform field 854B. When U=0 and the MOD field 742 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 852 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 852B and the beta field 854 (EVEX byte 3, bits [6:4]– SSS) is interpreted as a three bit data manipulation field 854C.

[0129] When U=1, the alpha field 852 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 852C. When U=1 and the MOD field 742 contains 11 (signifying a no memory access operation), part of the beta field 854 (EVEX byte 3, bit [4]- S₀) is interpreted as the RL field 857A; when it contains a 1 (round 857A.1) the rest of the beta field 854 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the round operation field 859A, while when the RL field 857A contains a 0 (VSIZE 857.A2) the rest of the beta field 854 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the vector length field 859B (EVEX byte 3, bit [6-5]- L₁₋₀). When U=1 and the MOD field 742 contains 00, 01, or 10 (signifying a memory access operation), the beta field 854 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 859B (EVEX byte 3, bit [6-5]- L₁₋₀) and the broadcast field 857B (EVEX byte 3, bit [4]- B).

Exemplary Register Architecture

[0130] **Figure 8** is a block diagram of a register architecture 800 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 810 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format 700 operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 859B	A (Figure 6A; U=0)	610, 615, 625, 630	zmm registers (the vector length is 64 byte)
	B (Figure 6B; U=1)	612	zmm registers (the vector length is 64 byte)
Instruction templates that do include the	B (Figure 6B; U=1)	617, 627	zmm, ymm, or xmm registers (the vector length is 64 bytes, 32 bytes,

vector length field 659B			or 16 bytes) depending on the vector length field 859B
-----------------------------	--	--	---

[0131] In other words, the vector length field 859B selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length; and instructions templates without the vector length field 859B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 700 operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

[0132] Write mask registers 815 - in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 815 are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

[0133] General-purpose registers 825 - in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

[0134] Scalar floating point stack register file (x87 stack) 845, on which is aliased the MMX packed integer flat register file 850 - in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

[0135] Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Exemplary Core Architectures, Processors, and Computer Architectures

[0136] Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures.

Exemplary Core Architectures

In-order and out-of-order core block diagram

[0137] **Figure 9A** is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. **Figure 9B** is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in **Figures 9A-9B** illustrate the in-order

pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

[0138] In **Figure 9A**, a processor pipeline 900 includes a fetch stage 902, a length decode stage 904, a decode stage 906, an allocation stage 908, a renaming stage 910, a scheduling (also known as a dispatch or issue) stage 912, a register read/memory read stage 914, an execute stage 916, a write back/memory write stage 918, an exception handling stage 922, and a commit stage 924.

[0139] **Figure 9B** shows processor core 990 including a front end unit 930 coupled to an execution engine unit 950, and both are coupled to a memory unit 970. The core 990 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 990 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

[0140] The front end unit 930 includes a branch prediction unit 932 coupled to an instruction cache unit 934, which is coupled to an instruction translation lookaside buffer (TLB) 936, which is coupled to an instruction fetch unit 938, which is coupled to a decode unit 940. The decode unit 940 (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 940 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 990 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 940 or otherwise within the front end unit 930). The decode unit 940 is coupled to a rename/allocator unit 952 in the execution engine unit 950.

[0141] The execution engine unit 950 includes the rename/allocator unit 952 coupled to a retirement unit 954 and a set of one or more scheduler unit(s) 956. The scheduler unit(s) 956 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 956 is coupled to the physical register file(s) unit(s) 958. Each of the physical register file(s) units 958 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point,, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit 958 comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) 958 is overlapped by the retirement unit 954 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit 954 and the physical register file(s) unit(s) 958 are coupled to the execution cluster(s) 960. The execution cluster(s) 960 includes a set of one or more execution units 962 and a set of one or more memory access units 964. The execution units 962 may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) 956, physical register file(s) unit(s) 958, and execution cluster(s) 960 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments are implemented in which only

the execution cluster of this pipeline has the memory access unit(s) 964). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

[0142] The set of memory access units 964 is coupled to the memory unit 970, which includes a data TLB unit 972 coupled to a data cache unit 974 coupled to a level 2 (L2) cache unit 976. In one exemplary embodiment, the memory access units 964 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 972 in the memory unit 970. The instruction cache unit 934 is further coupled to a level 2 (L2) cache unit 976 in the memory unit 970. The L2 cache unit 976 is coupled to one or more other levels of cache and eventually to a main memory.

[0143] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 900 as follows: 1) the instruction fetch unit 938 performs the fetch and length decoding stages 902 and 904; 2) the decode unit 940 performs the decode stage 906; 3) the rename/allocator unit 952 performs the allocation stage 908 and renaming stage 910; 4) the scheduler unit(s) 956 performs the schedule stage 912; 5) the physical register file(s) unit(s) 958 and the memory unit 970 perform the register read/memory read stage 914; the execution cluster 960 perform the execute stage 916; 6) the memory unit 970 and the physical register file(s) unit(s) 958 perform the write back/memory write stage 918; 7) various units may be involved in the exception handling stage 922; and 8) the retirement unit 954 and the physical register file(s) unit(s) 958 perform the commit stage 924.

[0144] The core 990 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 990 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[0145] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including

time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

[0146] While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units 934/974 and a shared L2 cache unit 976, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Specific Exemplary In-Order Core Architecture

[0147] Figures 10A-10B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

[0148] Figure 10A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1002 and with its local subset of the Level 2 (L2) cache 1004, according to embodiments of the invention. In one embodiment, an instruction decoder 1000 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1006 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1008 and a vector unit 1010 use separate register sets (respectively, scalar registers 1012 and vector registers 1014) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1006, alternative embodiments of the invention may use a

different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

[0149] The local subset of the L2 cache 1004 is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache 1004. Data read by a processor core is stored in its L2 cache subset 1004 and can be accessed quickly, in parallel with other processor cores accessing their own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset 1004 and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

[0150] **Figure 10B** is an expanded view of part of the processor core in **Figure 10A** according to embodiments of the invention. **Figure 10B** includes an L1 data cache 1006A part of the L1 cache 1004, as well as more detail regarding the vector unit 1010 and the vector registers 1014. Specifically, the vector unit 1010 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1028), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit 1020, numeric conversion with numeric convert units 1022A-B, and replication with replication unit 1024 on the memory input. Write mask registers 1026 allow predicating resulting vector writes.

[0151] **Figure 11** is a block diagram of a processor 1100 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in **Figure 11** illustrate a processor 1100 with a single core 1102A, a system agent 1110, a set of one or more bus controller units 1116, while the optional addition of the dashed lined boxes illustrates an alternative processor 1100 with multiple cores 1102A-N, a set of one or more integrated memory controller unit(s) 1114 in the system agent unit 1110, and special purpose logic 1108.

[0152] Thus, different implementations of the processor 1100 may include: 1) a CPU with the special purpose logic 1108 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 1102A-N being one or more general

purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores 1102A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 1102A-N being a large number of general purpose in-order cores. Thus, the processor 1100 may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor 1100 may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

[0153] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 1106, and external memory (not shown) coupled to the set of integrated memory controller units 1114. The set of shared cache units 1106 may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit 1112 interconnects the integrated graphics logic 1108 (integrated graphics logic 1108 is an example of and is also referred to herein as special purpose logic), the set of shared cache units 1106, and the system agent unit 1110/integrated memory controller unit(s) 1114, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units 1106 and cores 1102-A-N.

[0154] In some embodiments, one or more of the cores 1102A-N are capable of multi-threading. The system agent 1110 includes those components coordinating and operating cores 1102A-N. The system agent unit 1110 may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores 1102A-N and the integrated graphics logic 1108. The display unit is for driving one or more externally connected displays.

[0155] The cores 1102A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 1102A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Computer Architectures

[0156] **Figures 12-15** are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

[0157] Referring now to **Figure 12**, shown is a block diagram of a system 1200 in accordance with one embodiment of the present invention. The system 1200 may include one or more processors 1210, 1215, which are coupled to a controller hub 1220. In one embodiment, the controller hub 1220 includes a graphics memory controller hub (GMCH) 1290 and an Input/Output Hub (IOH) 1250 (which may be on separate chips); the GMCH 1290 includes memory and graphics controllers to which are coupled memory 1240 and a coprocessor 1245; the IOH 1250 couples input/output (I/O) devices 1260 to the GMCH 1290. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 1240 and the coprocessor 1245 are coupled directly to the processor 1210, and the controller hub 1220 in a single chip with the IOH 1250.

[0158] The optional nature of additional processors 1215 is denoted in **Figure 12** with broken lines. Each processor 1210, 1215 may include one or more of the processing cores described herein and may be some version of the processor 1100.

[0159] The memory 1240 may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment,

the controller hub 1220 communicates with the processor(s) 1210, 1215 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 1295.

[0160] In one embodiment, the coprocessor 1245 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub 1220 may include an integrated graphics accelerator.

[0161] There can be a variety of differences between the physical resources 1210, 1215 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[0162] In one embodiment, the processor 1210 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 1210 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 1245. Accordingly, the processor 1210 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 1245. Coprocessor(s) 1245 accept and execute the received coprocessor instructions.

[0163] Referring now to **Figure 13**, shown is a block diagram of a first more specific exemplary system 1300 in accordance with an embodiment of the present invention. As shown in **Figure 13**, multiprocessor system 1300 is a point-to-point interconnect system, and includes a first processor 1370 and a second processor 1380 coupled via a point-to-point interconnect 1350. Each of processors 1370 and 1380 may be some version of the processor 1100. In one embodiment of the invention, processors 1370 and 1380 are respectively processors 1210 and 1215, while coprocessor 1338 is coprocessor 1245. In another embodiment, processors 1370 and 1380 are respectively processor 1210 coprocessor 1245.

[0164] Processors 1370 and 1380 are shown including integrated memory controller (IMC) units 1372 and 1382, respectively. Processor 1370 also includes as part of its bus controller units point-to-point (P-P) interfaces 1376 and 1378; similarly, second processor 1380 includes P-P interfaces 1386 and 1388. Processors 1370, 1380 may exchange information via

a point-to-point (P-P) interface 1350 using P-P interface circuits 1378, 1388. As shown in **Figure 13**, IMCs 1372 and 1382 couple the processors to respective memories, namely a memory 1332 and a memory 1334, which may be portions of main memory locally attached to the respective processors.

[0165] Processors 1370, 1380 may each exchange information with a chipset 1390 via individual P-P interfaces 1352, 1354 using point to point interface circuits 1376, 1394, 1386, 1398. Chipset 1390 may optionally exchange information with the coprocessor 1338 via a high-performance interface 1392. In one embodiment, the coprocessor 1338 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

[0166] A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

[0167] Chipset 1390 may be coupled to a first bus 1316 via an interface 1396. In one embodiment, first bus 1316 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present invention is not so limited.

[0168] As shown in **Figure 13**, various I/O devices 1314 may be coupled to first bus 1316, along with a bus bridge 1318 which couples first bus 1316 to a second bus 1320. In one embodiment, one or more additional processor(s) 1315, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 1316. In one embodiment, second bus 1320 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 1320 including, for example, a keyboard and/or mouse 1322, communication devices 1327 and a storage unit 1328 such as a disk drive or other mass storage device which may include instructions/code and data 1330, in one embodiment. Further, an audio I/O 1324 may be coupled to the second bus 1320. Note that other architectures are possible. For example,

instead of the point-to-point architecture of **Figure 13**, a system may implement a multi-drop bus or other such architecture.

[0169] Referring now to **Figure 14**, shown is a block diagram of a second more specific exemplary system 1400 in accordance with an embodiment of the present invention. Like elements in **Figures 13 and 14** bear like reference numerals, and certain aspects of **Figure 13** have been omitted from **Figure 14** in order to avoid obscuring other aspects of **Figure 14**.

[0170] **Figure 14** illustrates that the processors 1370, 1380 may include integrated memory and I/O control logic ("CL") 1372 and 1382, respectively. Thus, the CL 1372, 1382 include integrated memory controller units and include I/O control logic. **Figure 12** illustrates that not only are the memories 1332, 1334 coupled to the CL 1372, 1382, but also that I/O devices 1414 are also coupled to the control logic 1372, 1382. Legacy I/O devices 1415 are coupled to the chipset 1390.

[0171] Referring now to **Figure 15**, shown is a block diagram of a SoC 1500 in accordance with an embodiment of the present invention. Similar elements in **Figure 11** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 15**, an interconnect unit(s) 1502 is coupled to: an application processor 1510 which includes a set of one or more cores 1102A-N, which include cache units 1104A-N, and shared cache unit(s) 1106; a system agent unit 1110; a bus controller unit(s) 1116; an integrated memory controller unit(s) 1114; a set or one or more coprocessors 1520 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 1530; a direct memory access (DMA) unit 1532; and a display unit 1540 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 1520 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

[0172] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage

system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[0173] Program code, such as code 1330 illustrated in **Figure 13**, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

[0174] The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

[0175] One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as “IP cores” may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[0176] Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

[0177] Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

Emulation (including binary translation, code morphing, etc.)

[0178] In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

[0179] **Figure 16** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. **Figure 16** shows a program in a high level language 1602 may be compiled using an x86 compiler 1604 to generate x86 binary code 1606 that may be natively executed by a processor with at least one x86 instruction set core 1616. The processor with at least one x86 instruction set core 1616 represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler 1604 represents a compiler that is operable to generate x86 binary code 1606 (e.g., object code) that can, with or without additional

linkage processing, be executed on the processor with at least one x86 instruction set core 1616. Similarly, **Figure 16** shows the program in the high level language 1602 may be compiled using an alternative instruction set compiler 1608 to generate alternative instruction set binary code 1610 that may be natively executed by a processor without at least one x86 instruction set core 1614 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). The instruction converter 1612 is used to convert the x86 binary code 1606 into code that may be natively executed by the processor without an x86 instruction set core 1614. This converted code is not likely to be the same as the alternative instruction set binary code 1610 because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter 1612 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code 1606.

FURTHER EXAMPLES

[0180] Example 1 provides a method of executing an instruction, the method comprising: decoding an instruction by a decode circuit, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and executing the decoded instruction by an execution circuit by: for each data element position of a source vector identified by the source vector identifier, determining a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and storing, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0181] **Example 2** includes the substance of the exemplary method of **Example 1**, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

[0182] **Example 3** includes the substance of the exemplary method of **Example 1**, wherein the destination vector identifier and the source vector identifier identify storage locations.

[0183] **Example 4** includes the substance of the exemplary method of **Example 1**, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

[0184] **Example 5** includes the substance of the exemplary method of **Example 1**, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value storing the value representing the number of matching data element positions further includes adding the value to a corresponding data element position of a second source vector identified by the second source vector identifier.

[0185] **Example 6** provides a system for executing an instruction, the system comprising: a decode circuit to decode an instruction, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and an execution circuit to execute the decoded instruction to, for each data element position of a source vector identified by the source vector identifier, determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0186] **Example 7** includes the substance of the exemplary system of **Example 6**, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

[0187] **Example 8** includes the substance of the exemplary system of **Example 6**, wherein the destination vector identifier and the source vector identifier identify storage locations.

[0188] **Example 9** includes the substance of the exemplary system of **Example 6**, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

[0189] **Example 10** includes the substance of the exemplary system of **Example 6**, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

[0190] **Example 11** provides a processor for executing an instruction, the processor comprising: a decode circuit to decode an instruction, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and an execution circuit to execute the decoded instruction to, for each data element position of a source vector identified by the source vector identifier, determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0191] **Example 12** includes the substance of the exemplary processor of **Example 11**, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

[0192] **Example 13** includes the substance of the exemplary processor of **Example 11**, wherein the destination vector identifier and the source vector identifier identify storage locations.

[0193] **Example 14** includes the substance of the exemplary processor of **Example 11**, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

[0194] Example 15 includes the substance of the exemplary processor of **Example 11**, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

[0195] Example 16 provides an apparatus to execute an instruction, the apparatus comprising: means for decoding an instruction, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and means for executing the decoded instruction to, for each data element position of a source vector identified by the source vector identifier, determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0196] Example 17 includes the substance of the exemplary apparatus of **Example 16**, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

[0197] Example 18 includes the substance of the exemplary apparatus of **Example 16**, wherein the destination vector identifier and the source vector identifier identify storage locations.

[0198] Example 19 includes the substance of the exemplary apparatus of **Example 16**, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

[0199] Example 20 includes the substance of the exemplary apparatus of **Example 16**, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching

data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

[0200] Example 21 provides a non-transitory machine-readable medium containing instructions that, when performed by a processor, cause the performance of operations comprising: fetching an instruction from a code storage by a fetch circuit, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; decoding the fetched instruction by a decode circuit; and executing the decoded instruction by an execution circuit on a source vector identified by the source vector identifier to, for each data element position of the source vector, determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

[0201] Example 22 includes the substance of the exemplary non-transitory machine-readable medium of **Example 21**, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

[0202] Example 23 includes the substance of the exemplary non-transitory machine-readable medium of **Example 21**, wherein the destination vector identifier and the source vector identifier identify storage locations.

[0203] Example 24 includes the substance of the exemplary non-transitory machine-readable medium of **Example 21**, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

[0204] Example 25 includes the substance of the exemplary non-transitory machine-readable medium of **Example 21**, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position

of a second source vector identified by the second source vector identifier, and the immediate value.

CLAIMS

What is claimed is:

1. A method for executing an instruction, the method comprising:
decoding an instruction by a decode circuit, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and
executing the decoded instruction by an execution circuit by:
for each data element position of a source vector identified by the source vector identifier,
determining a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and
storing, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.
2. The method of claim 1, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.
3. The method of claim 1, wherein the destination vector identifier and the source vector identifier identify storage locations.
4. The method of claim 1, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.
5. The method of claim 1, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further

comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value storing the value representing the number of matching data element positions further includes adding the value to a corresponding data element position of a second source vector identified by the second source vector identifier.

6. A processor for executing an instruction, the processor comprising:
a decode circuit to decode an instruction, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and
an execution circuit to execute the decoded instruction to, for each data element position of a source vector identified by the source vector identifier,
determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and
store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

7. The processor of claim 6, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

8. The processor of claim 6, wherein the destination vector identifier and the source vector identifier identify storage locations.

9. The processor of claim 6, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

10. The processor of claim 6, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

11. An apparatus to execute an instruction, the apparatus comprising:
means for decoding an instruction, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value; and
means for executing the decoded instruction to, for each data element position of a source vector identified by the source vector identifier,
determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and
store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

12. The apparatus of claim 11, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

13. The apparatus of claim 11, wherein the destination vector identifier and the source vector identifier identify storage locations.

14. The apparatus of claim 11, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

15. The apparatus of claim 11, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

16. A non-transitory machine-readable medium containing instructions that, when performed by a processor, cause the performance of operations comprising:

fetching an instruction from a code storage by a fetch circuit, the instruction comprising an opcode, a destination vector identifier, a source vector identifier, and an immediate value;

decoding the fetched instruction by a decode circuit; and

executing the decoded instruction by an execution circuit on a source vector identified by the source vector identifier to, for each data element position of the source vector,

determine a number of matching data element positions in the source vector storing a same data value as stored at the data element position, the matching data element positions located between the data element position and a least significant data element position of the source vector, and

store, in a corresponding data element position of a destination vector identified by the destination vector identifier, a value representing the number of matching data element positions.

17. The non-transitory machine-readable medium of claim 16, wherein the stored value representing the number of matching data element positions is a sum of the number of matching data element positions and the immediate value.

18. The non-transitory machine-readable medium of claim 16, wherein the destination vector identifier and the source vector identifier identify storage locations.

19. The non-transitory machine-readable medium of claim 16, wherein a size of data elements in the source vector identified by the source vector identifier is one of byte, word, doubleword, and quadword.

20. The non-transitory machine-readable medium of claim 16, wherein the source vector identifier is a first source vector identifier and the source vector is a first source vector, and wherein the instruction further comprises a second source vector identifier, and wherein the stored value represents a sum of the number of matching data element positions, a corresponding data element position of a second source vector identified by the second source vector identifier, and the immediate value.

FIG. 1

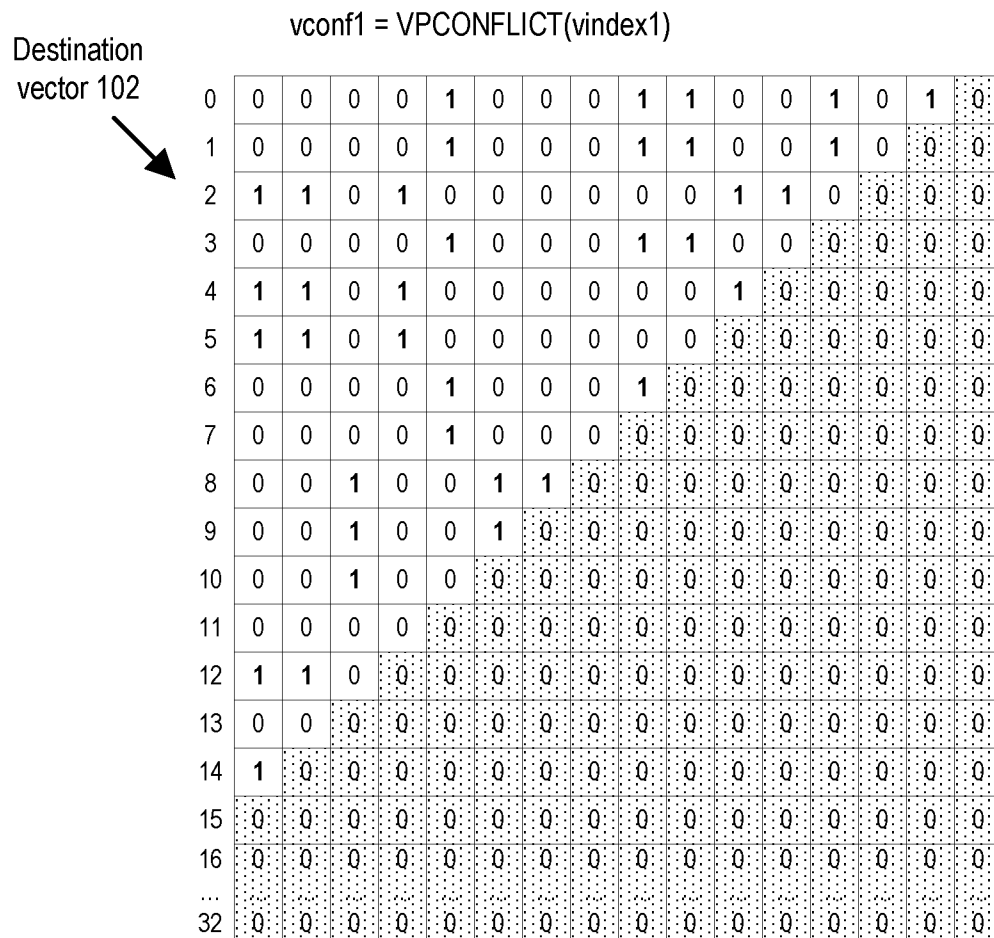
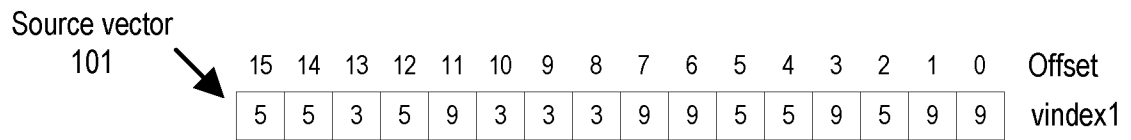


FIG. 2

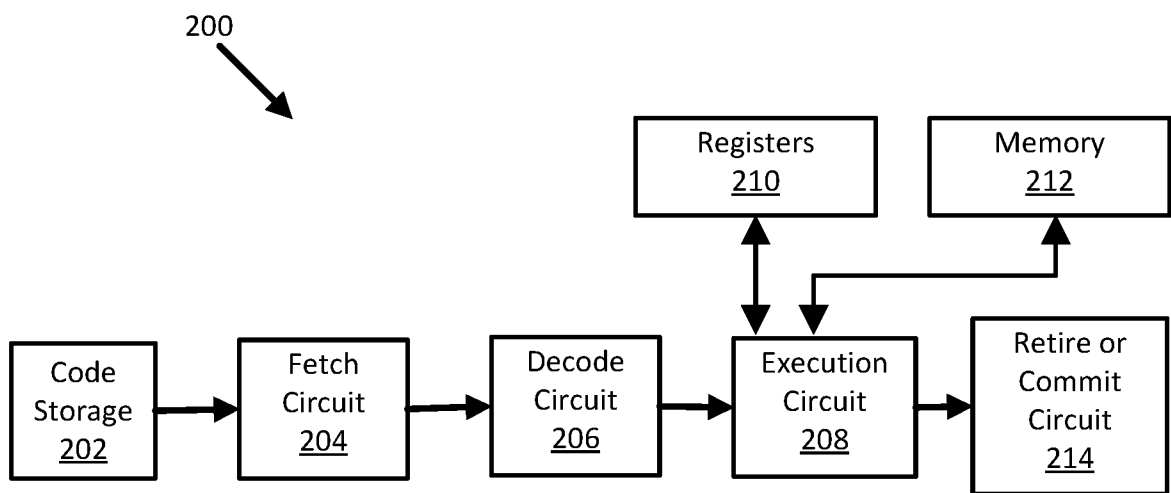


FIG. 3

Instruction 300A



OPCODE <u>302</u>	DESTINATION VECTOR IDENTIFIER <u>304</u>	SOURCE VECTOR IDENTIFIER <u>306</u>	IMMEDIATE VALUE <u>308</u>
----------------------	---	--	----------------------------------

Instruction 300B



OPCODE <u>310</u>	DESTINATION VECTOR IDENTIFIER <u>312</u>	FIRST SOURCE VECTOR IDENTIFIER <u>314</u>	SECOND SOURCE VECTOR IDENTIFIER <u>316</u>	IMMEDIATE VALUE <u>318</u>
----------------------	---	---	--	----------------------------------

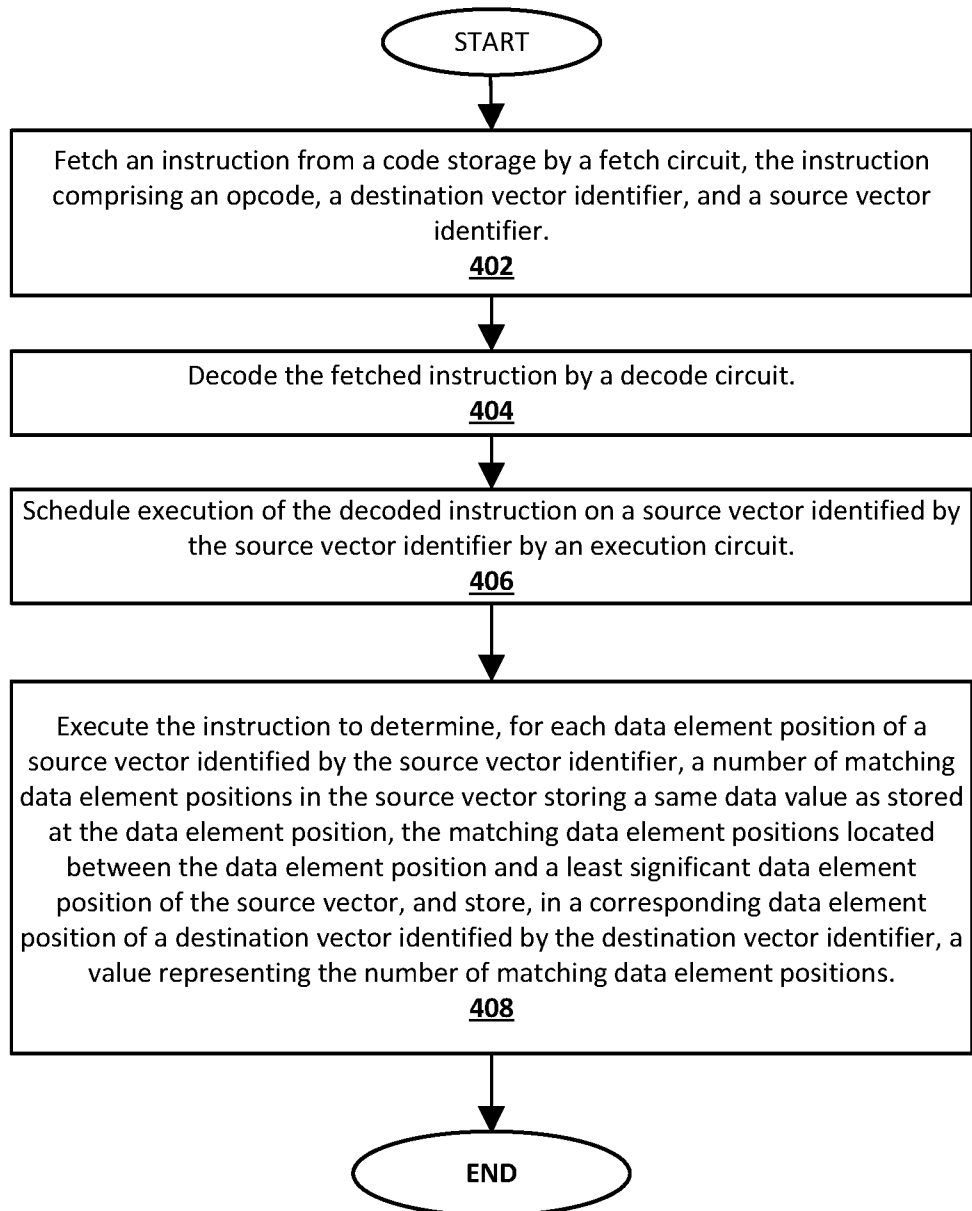
FIG. 4

FIG. 5A

500



Pseudocode for vector conflict and data value count instruction:

VCONF_CNT dest, src, imm8

```
for(i=0; i<KL; i++){  
    n = imm8;  
    for(k=0; k<i; k++){  
        if(src[k]==src[i]) n++;  
    }  
    dest[i] = n;  
}
```

FIG. 5B

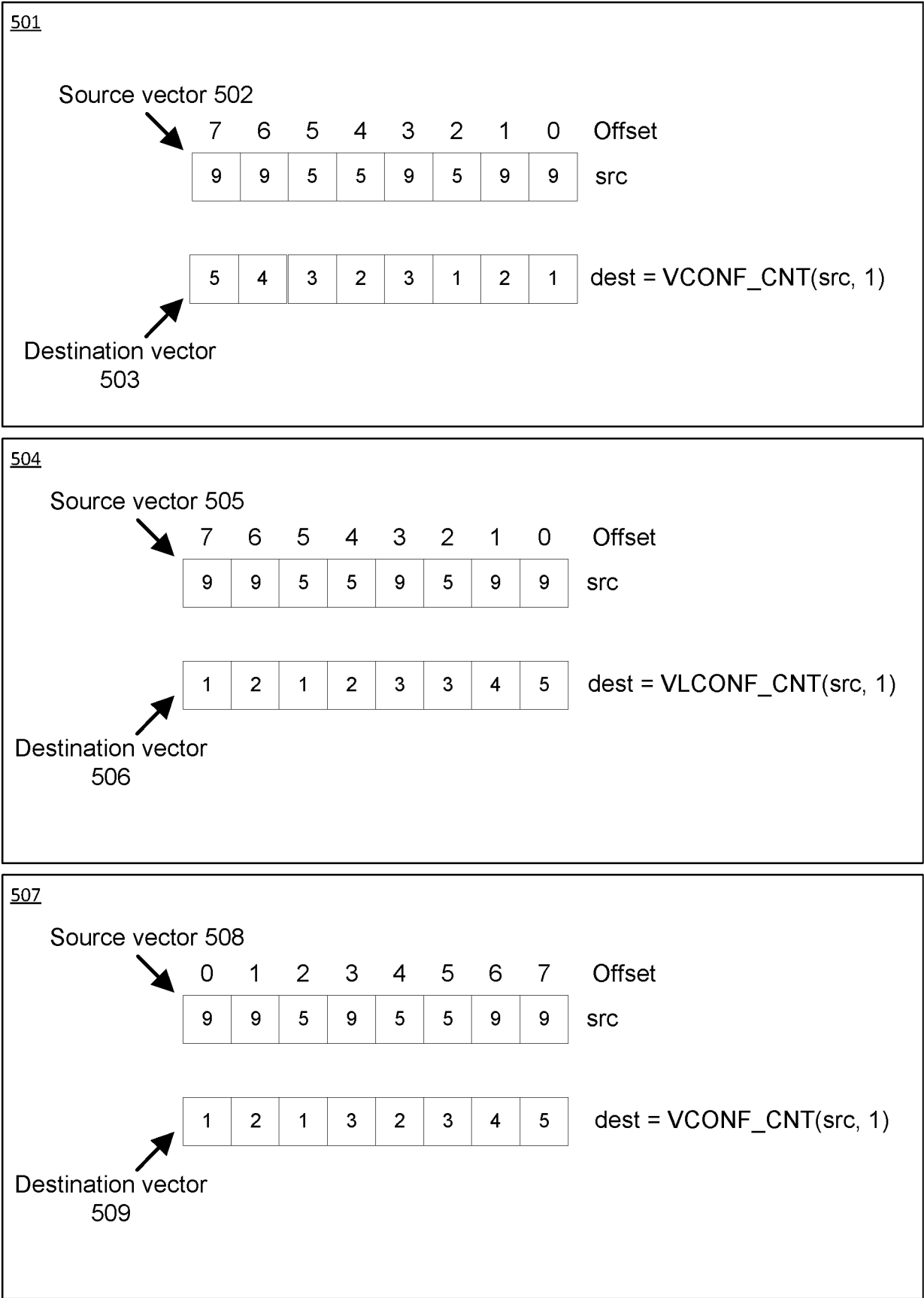


FIG. 5C

510



Pseudocode for alternate vector conflict and data value count instruction:

VCONF_CNT dest, src1, src2, imm8

```

for(i=0; i<KL; i++){
    n = src2[i] + imm8;
    for(k=0; k<i; k++){
        if(src1[k]==src1[i]) n++;
    }
    dest[i] = n;
}

```

511

First source vector

512



7	6	5	4	3	2	1	0	Offset
9	9	5	5	9	5	9	9	src1 //index for conflict check

Second source vector 513



A	A	B	B	A	B	A	A	src2 //result of GATHER
---	---	---	---	---	---	---	---	-------------------------

A+	A+	B+	B+	A+	B+	A+	A+	dest =
5	4	3	2	3	1	2	1	VCONF_CNT(src1,src2,1)

Destination vector 514



FIG. 6B

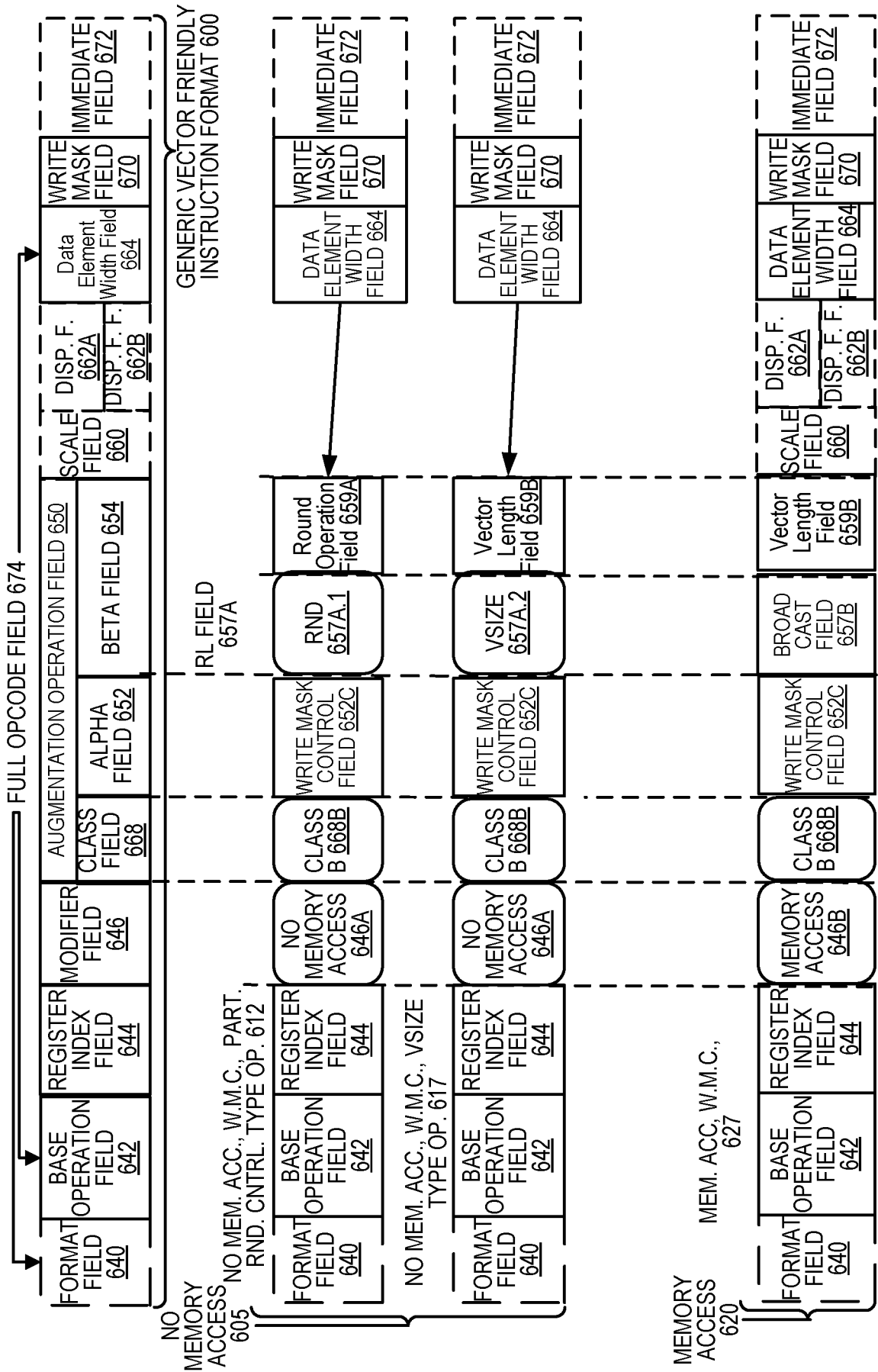


FIG. 7A

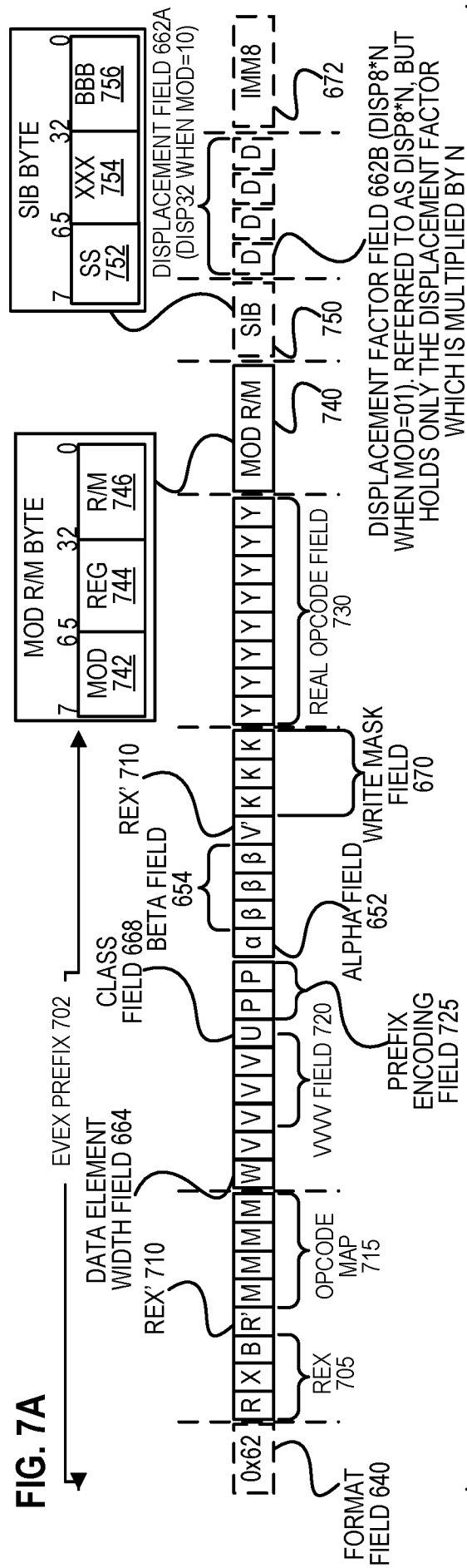


FIG. 7B

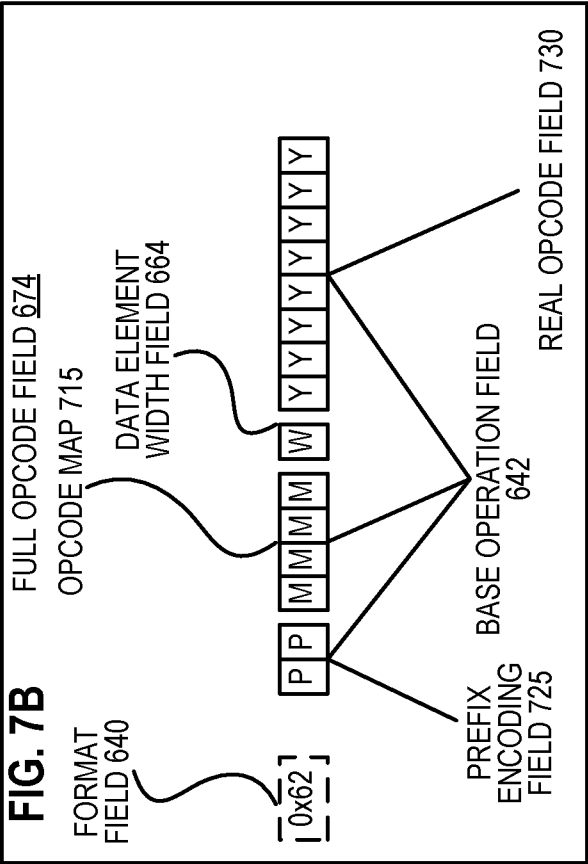


FIG. 7C

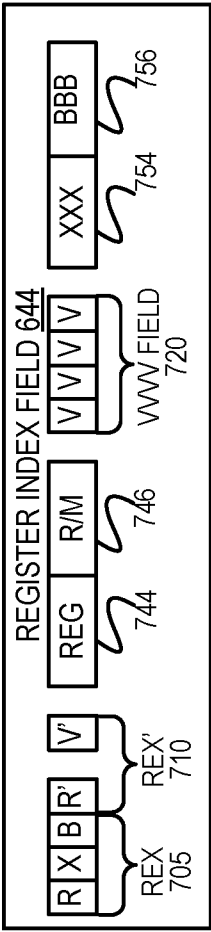


FIG. 7D

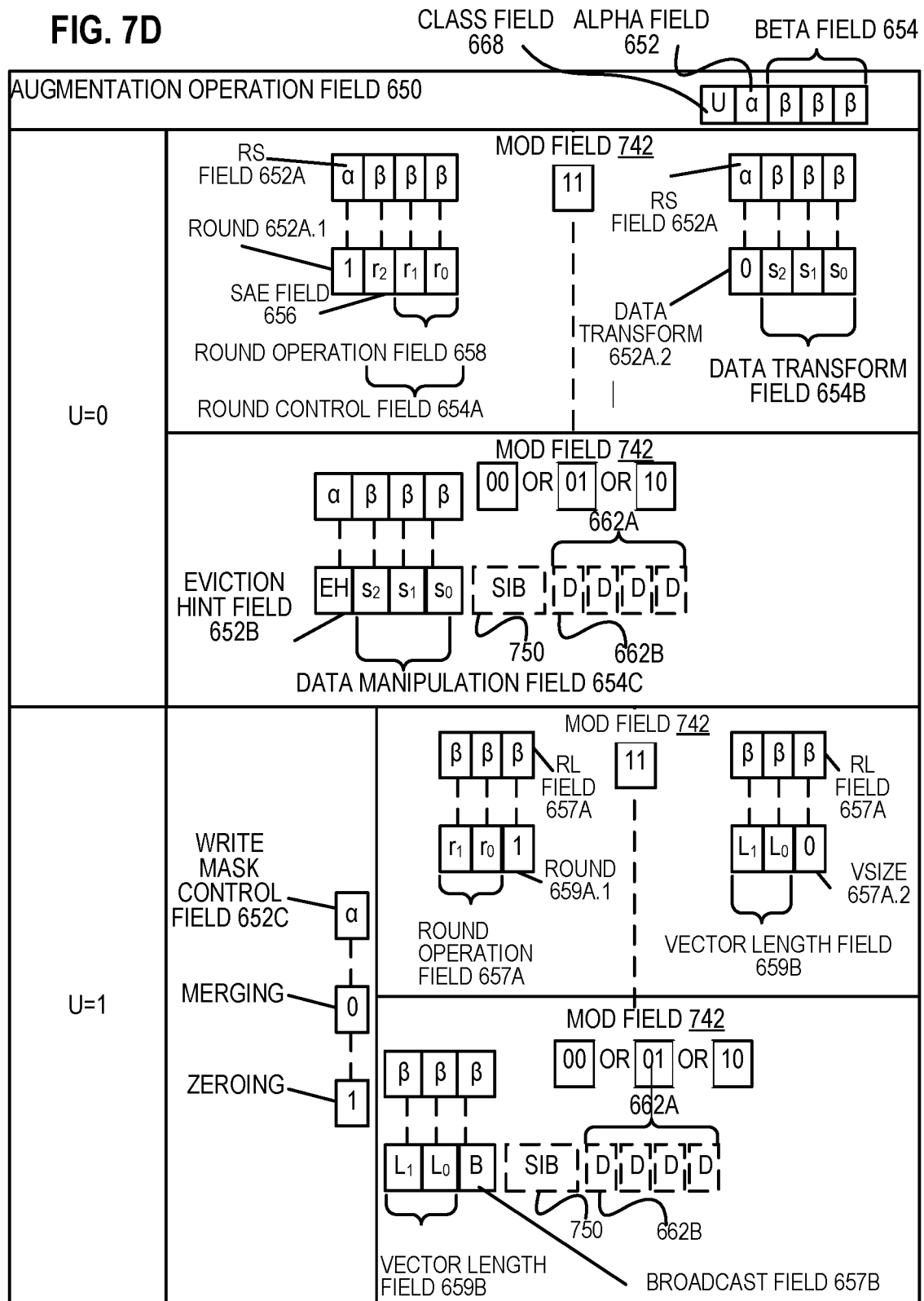


FIG. 8

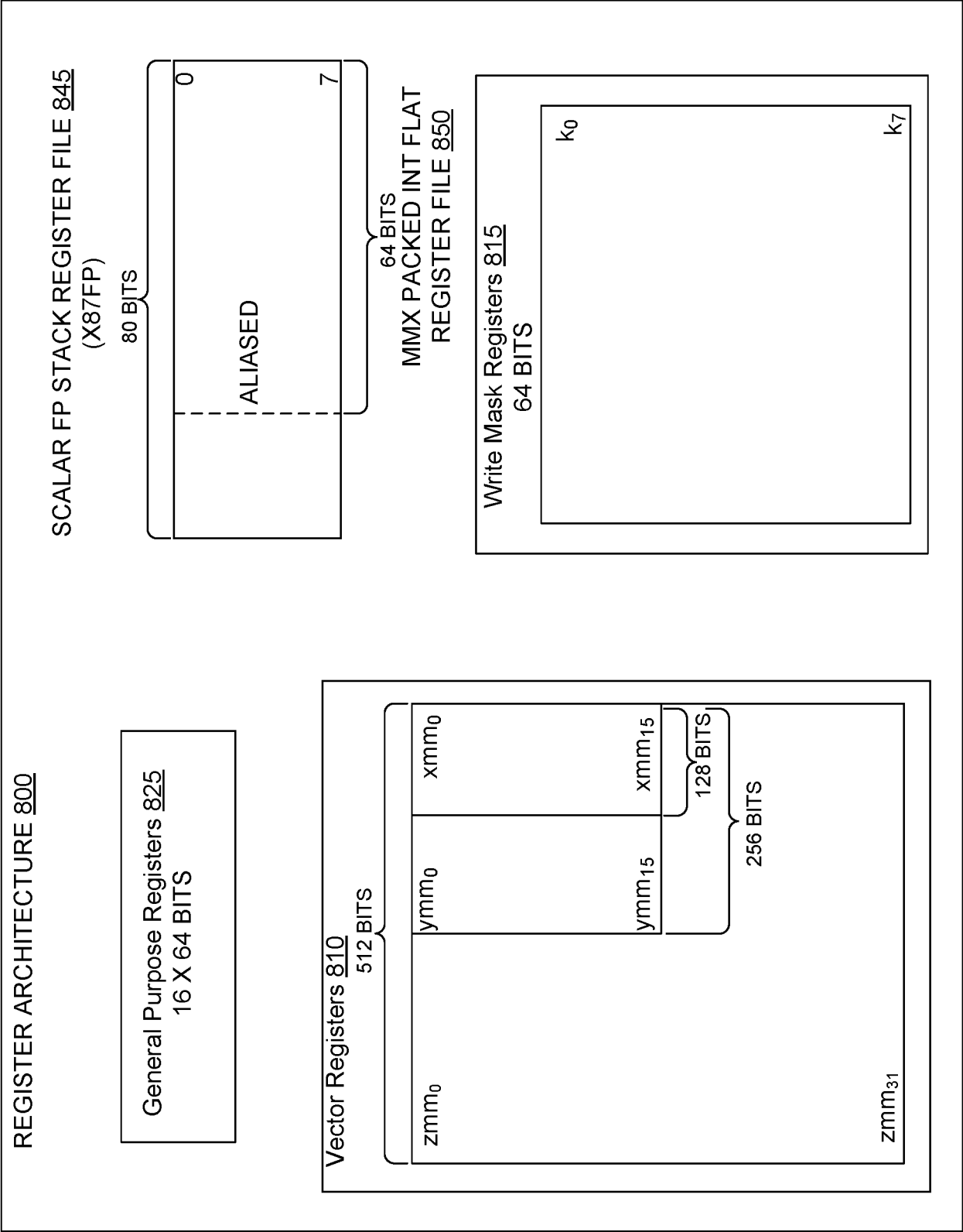




FIG. 9A

PIPELINE 900

CORE 990

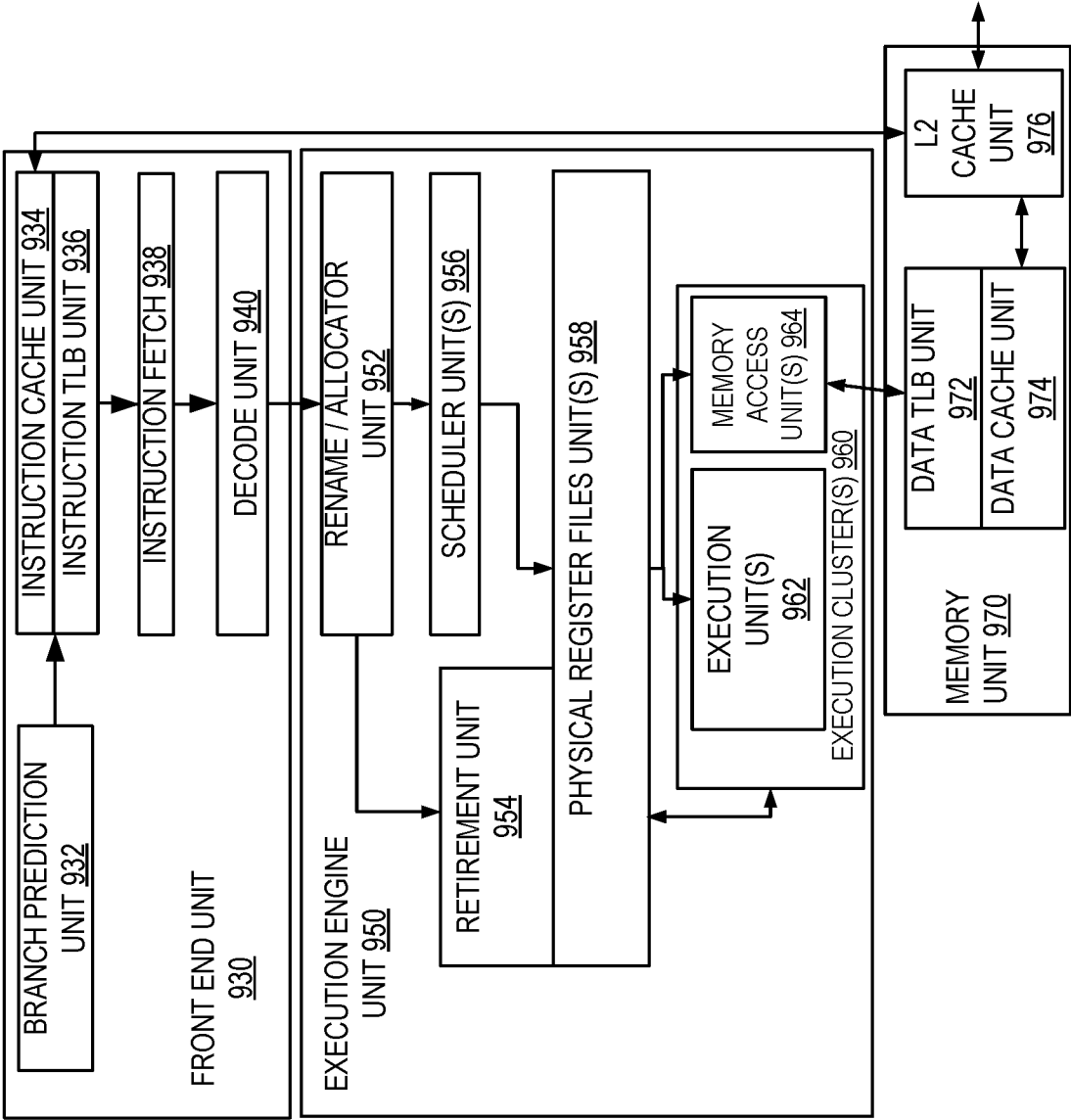


FIG. 9B

FIG. 10A

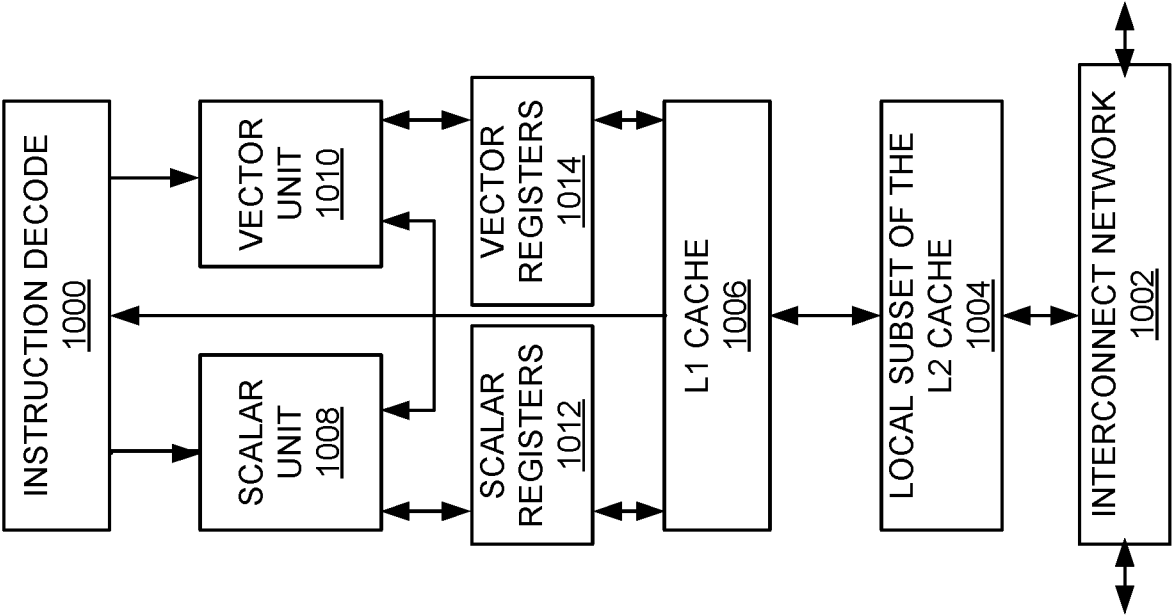
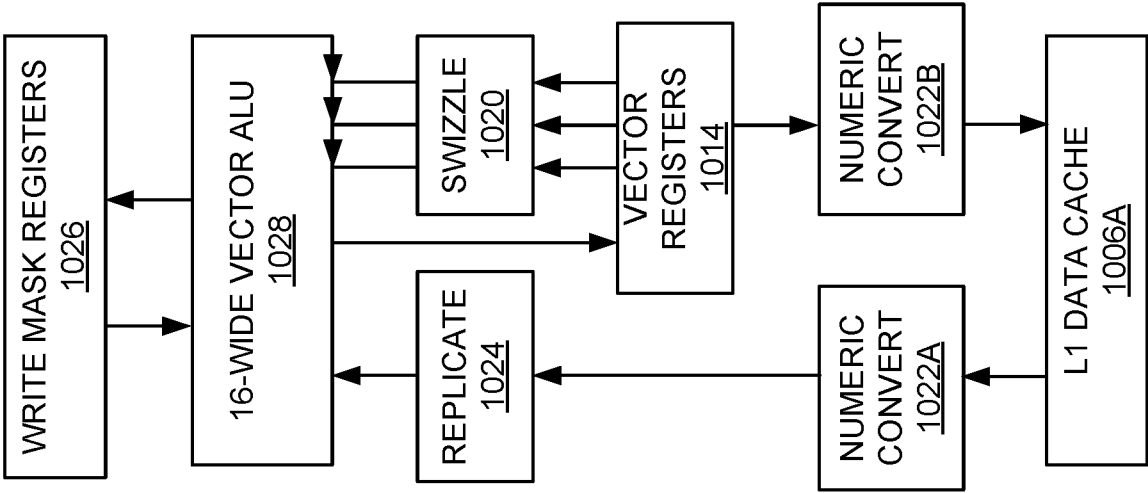
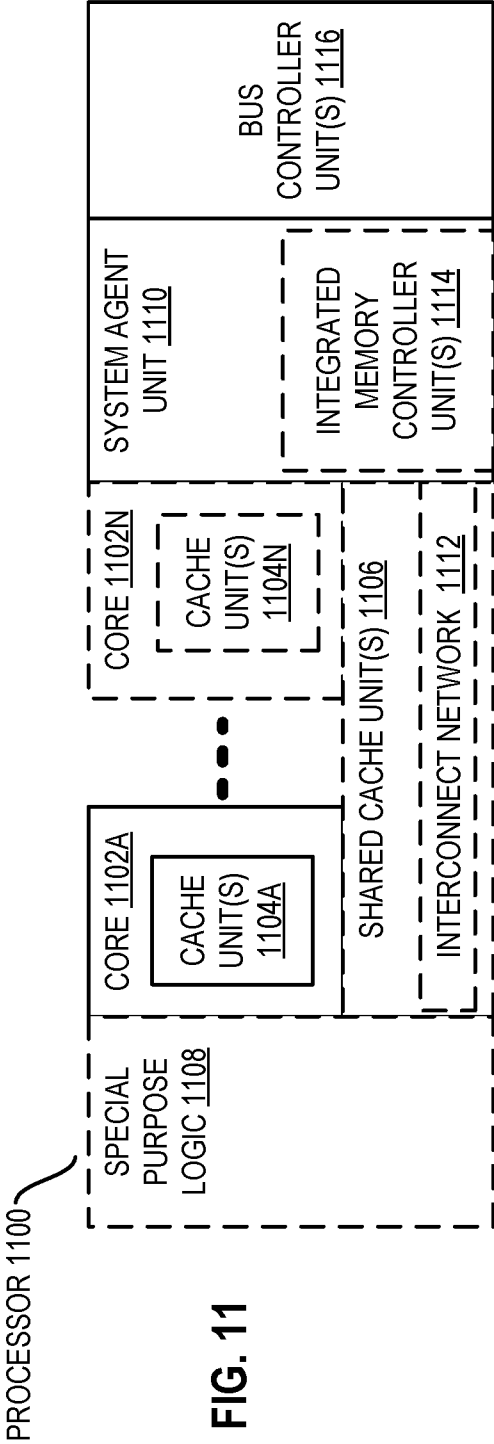
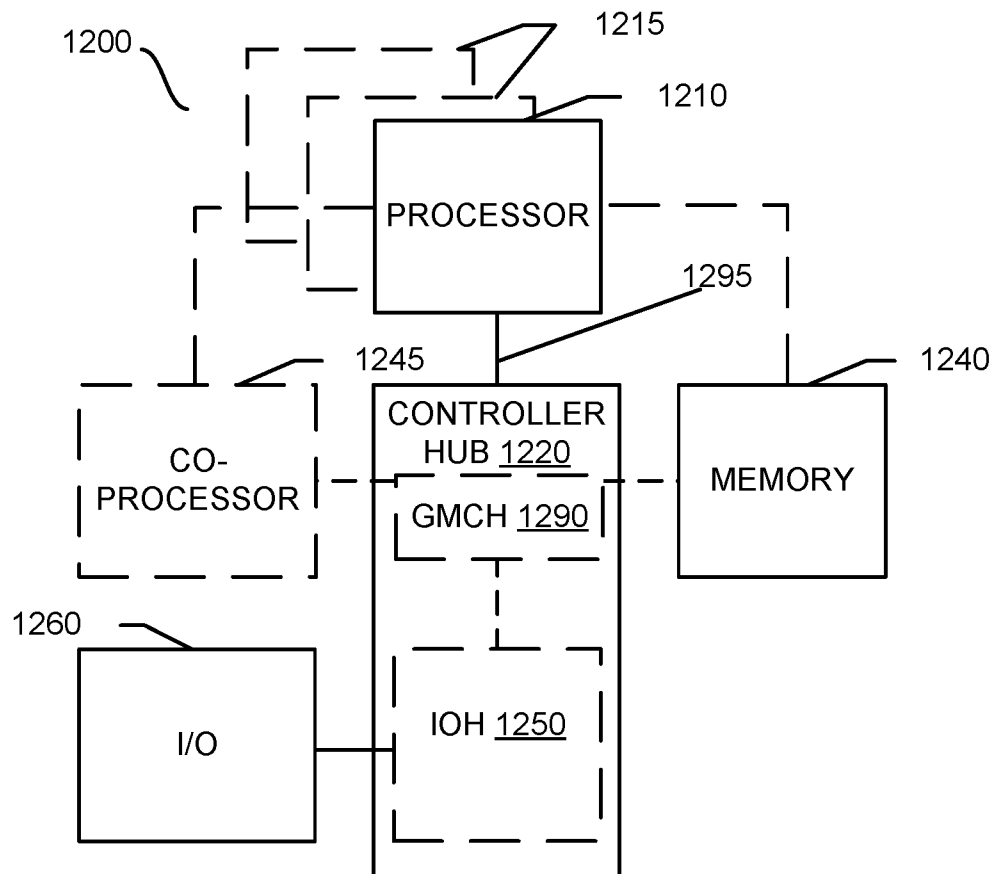


FIG. 10B





**FIG. 12**

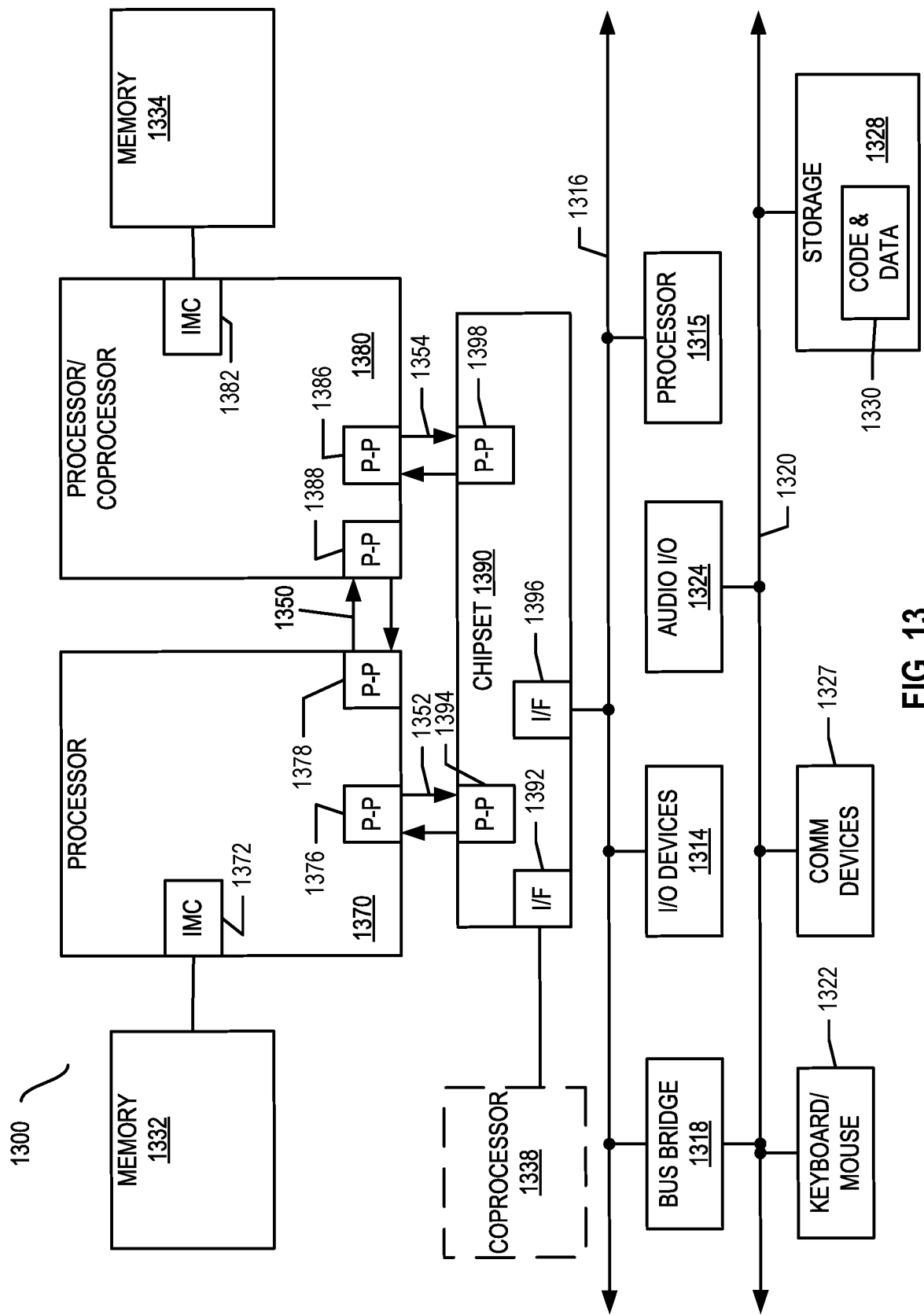


FIG. 13

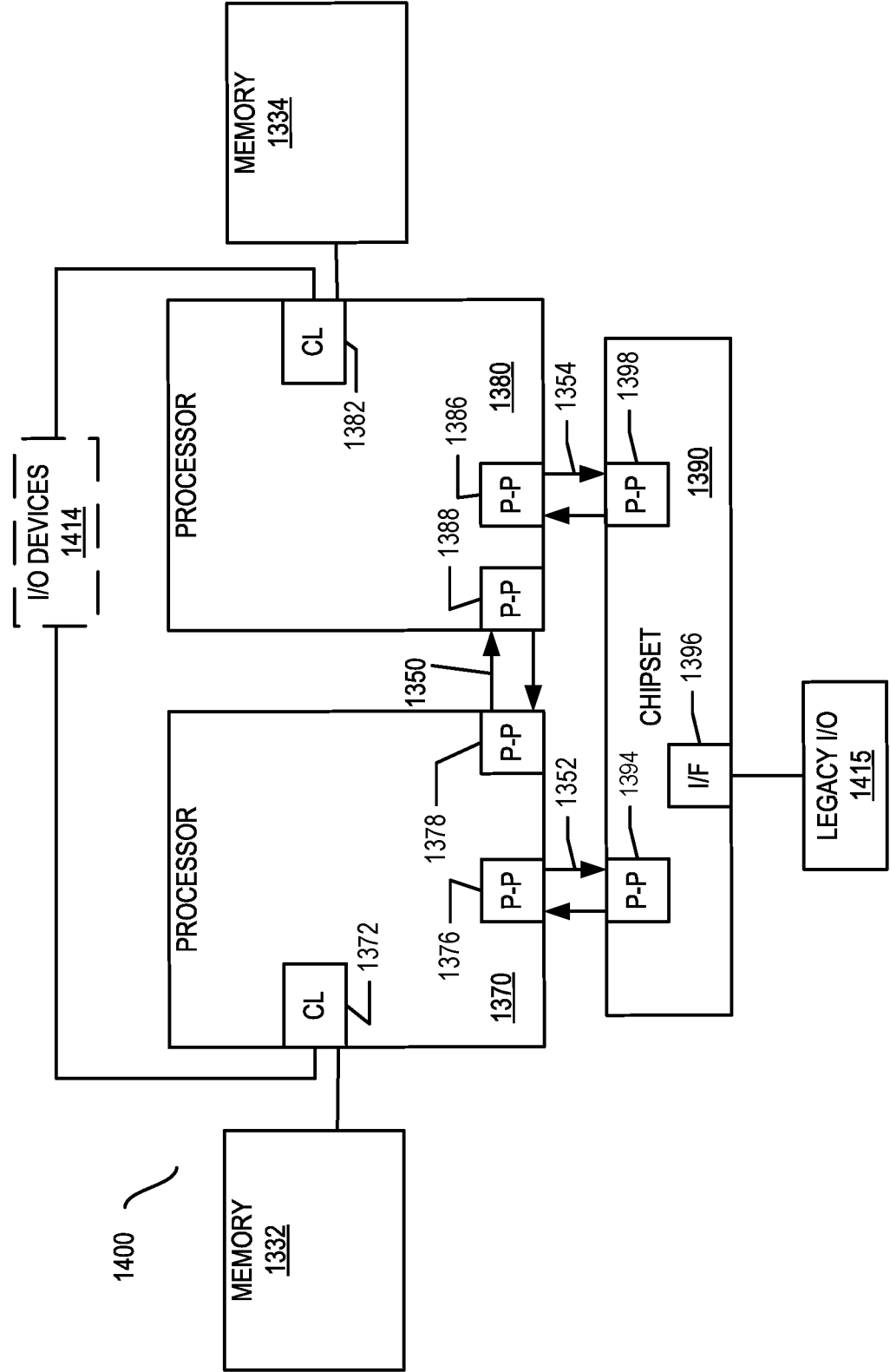


FIG. 14

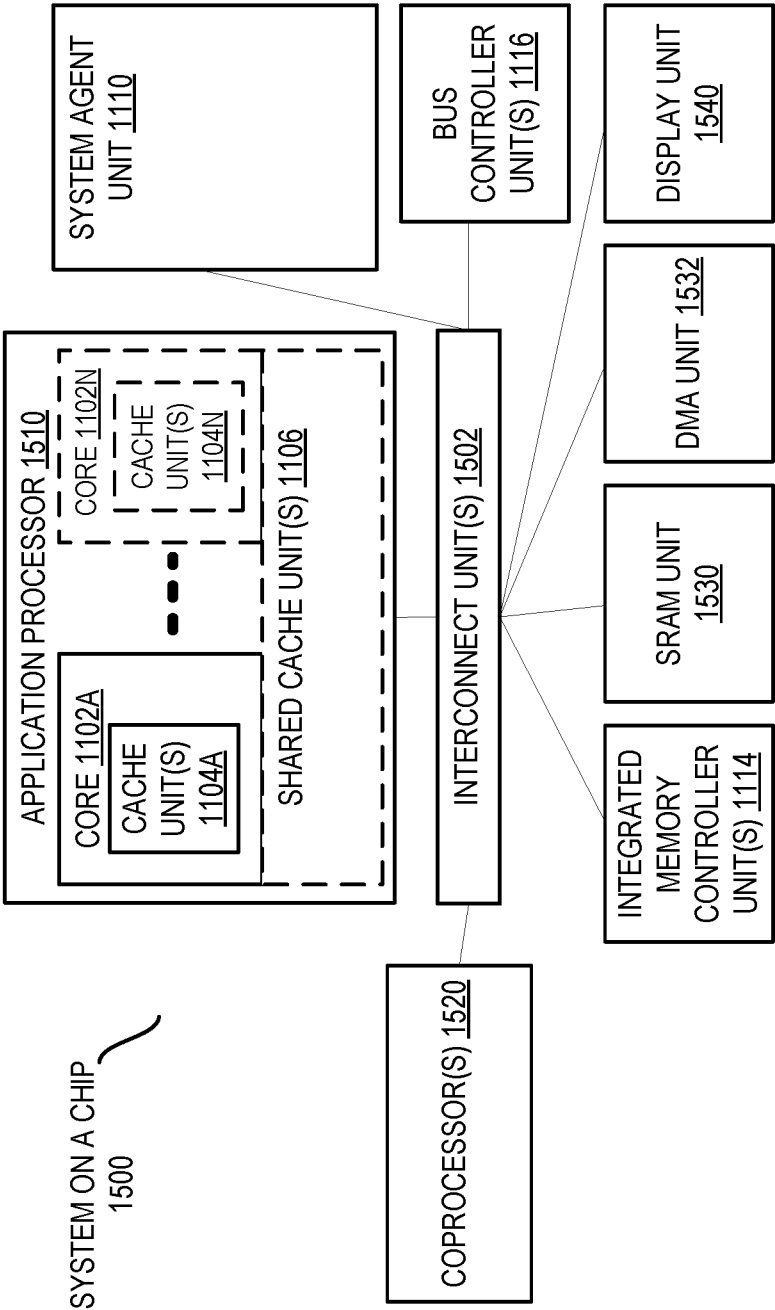
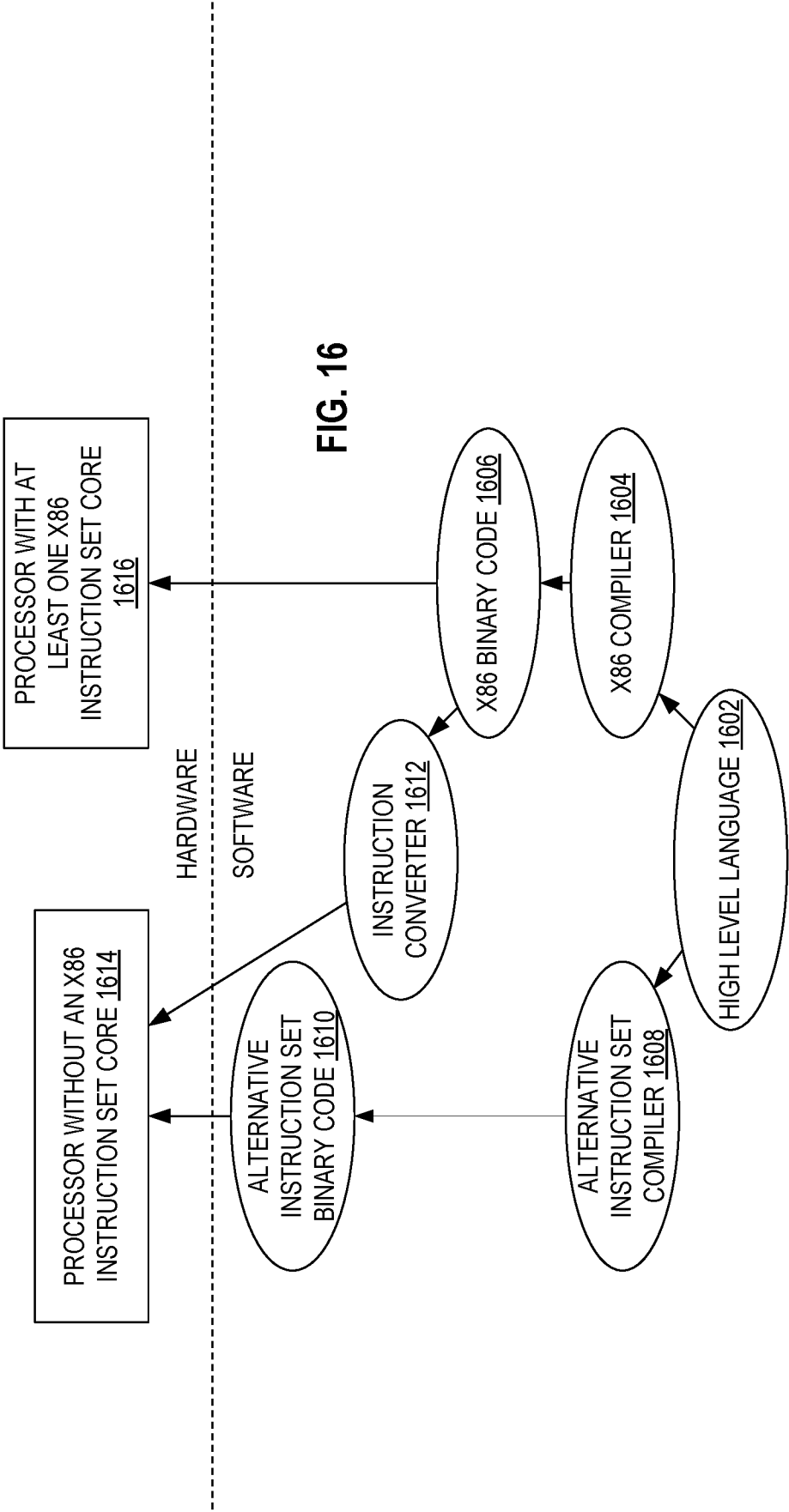


FIG. 15



INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/040509**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i**

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/30; G06F 9/02; G06F 17/16; G06F 9/06; G06F 9/00

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: source, destination, vector, immediate, determine, match, same, compare, count, conflict

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2014-0189294 A1 (MATT WALSH et al.) 03 July 2014 See paragraphs [0027], [0090]; claim 1; and figure 6.	1-20
A	US 2014-0108480 A1 (ELMOUSTAPHA OULD-AHMED-VALL et al.) 17 April 2014 See paragraphs [0006], [0068]; and figure 2.	1-20
A	US 2009-0100247 A1 (WILLIAM C. MOYER et al.) 16 April 2009 See paragraphs [0027], [0044]; and figure 1.	1-20
A	WO 2013-100989 A1 (INTEL CORPORATION et al.) 04 July 2013 See paragraphs [0006], [0035]; and figure 1.	1-20
A	US 2003-0135722 A1 (ANDREW JOHNSON) 17 July 2003 See paragraphs [0027], [0039]; and figure 7.	1-20



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search

20 February 2018 (20.02.2018)

Date of mailing of the international search report

20 February 2018 (20.02.2018)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

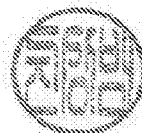


Facsimile No. +82-42-481-8578

Authorized officer

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/040509

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2014-0189294 A1	03/07/2014	None	
US 2014-0108480 A1	17/04/2014	CN 104011657 A CN 104011657 B TW 201331834 A TW 201723807 A TW I559220 B WO 2013-095592 A1	27/08/2014 12/10/2016 01/08/2013 01/07/2017 21/11/2016 27/06/2013
US 2009-0100247 A1	16/04/2009	US 8700884 B2	15/04/2014
WO 2013-100989 A1	04/07/2013	CN 104040482 A TW 201344565 A TW I517039 B US 2013-0339668 A1 US 2017-139715 A1 US 9557998 B2	10/09/2014 01/11/2013 11/01/2016 19/12/2013 18/05/2017 31/01/2017
US 2003-0135722 A1	17/07/2003	US 6918030 B2	12/07/2005